Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers

FUGA KAWAMATA, Waseda University, Japan
HIROSHI UNNO, University of Tsukuba, Japan
TARO SEKIYAMA, National Institute of Informatics, Japan
TACHIO TERAUCHI, Waseda University, Japan

Algebraic effects and handlers are a mechanism to structure programs with computational effects in a modular way. They are recently gaining popularity and being adopted in practical languages, such as OCaml. Meanwhile, there has been substantial progress in program verification via *refinement type systems*. While a variety of refinement type systems have been proposed, thus far there has not been a satisfactory refinement type system for algebraic effects and handlers. In this paper, we fill the void by proposing a novel refinement type system for languages with algebraic effects and handlers. The expressivity and usefulness of algebraic effects and handlers come from their ability to manipulate *delimited continuations*, but delimited continuations also complicate programs' control flow and make their verification harder. To address the complexity, we introduce a novel concept that we call *answer refinement modification* (ARM for short), which allows the refinement type system to precisely track what effects occur and in what order when a program is executed, and reflect such information as modifications to the refinements in the types of delimited continuations. We formalize our type system that supports ARM (as well as answer *type* modification, or ATM) and prove its soundness. Additionally, as a proof of concept, we have extended the refinement type system to a subset of OCaml 5 which comes with a built-in support for effect handlers, implemented a type checking and inference algorithm for the extension, and evaluated it on a number of benchmark programs that use algebraic effects and handlers. The evaluation demonstrates that ARM is conceptually simple and practically useful.

Finally, a natural alternative to directly reasoning about a program with delimited continuations is to apply a *continuation passing style* (CPS) transformation that transforms the program to a pure program without delimited continuations. We investigate this alternative in the paper, and show that the approach is indeed possible by proposing a novel CPS transformation for algebraic effects and handlers that enjoys bidirectional (refinement-)type-preservation. We show that there are pros and cons with this approach, namely, while one can use an existing refinement type checking and inference algorithm that can only (directly) handle pure programs, there are issues such as needing type annotations in source programs and making the inferred types less informative to a user.

CCS Concepts: • **Theory of computation → Type theory**; **Program verification**; **Control primitives**; • **Software and its engineering → Functional languages**; **Control structures**.

Additional Key Words and Phrases: algebraic effects and handlers, type-and-effect system, refinement type system, answer type modification, answer refinement modification, CPS transformation

Authors' addresses: Fuga Kawamata, Waseda University, Tokyo, Japan, maple-river@fuji.waseda.jp; Hiroshi Unno, University of Tsukuba, Tsukuba, Japan, uhiro@cs.tsukuba.ac.jp; Taro Sekiyama, National Institute of Informatics, Tokyo, Japan, ryukilon@gmail.com; Tachio Terauchi, Waseda University, Tokyo, Japan, terauchi@waseda.jp.

ACM 2475-1421/2024/1-ART5
https://doi.org/10.1145/3633280

ACM Reference Format:
Fuga Kawamata, Hiroshi Unno, Taro Sekiyama, and Tachio Terauchi. 2024. Answer Refinement Modification: Refinement Type System for Algebraic Effects and Handlers. *Proc. ACM Program. Lang.* 8, POPL, Article 5 (January 2024), 80 pages. https://doi.org/10.1145/3633280

1 INTRODUCTION

Algebraic effects [Plotkin and Power 2003] and handlers [Plotkin and Pretnar 2009, 2013] are a mechanism to structure programs with computational effects in a modular way. Algebraic effects represent abstracted computational effects and handlers specify their behaviors using delimited continuations. The ability to use delimited continuations makes algebraic effects and handlers highly expressive, allowing them to describe prominent computational effects such as exceptions, nondeterminism, mutable states, backtracking, and cooperative multithreading. Additionally, algebraic effects and handlers are recently gaining quite a recognition in practice and are adopted in popular programming languages, such as OCaml [Sivaramakrishnan et al. 2021].

Meanwhile, there has been substantial progress in program verification via *refinement type systems* [Bengtson et al. 2011; Freeman and Pfenning 1991; Nanjo et al. 2018; Rondon et al. 2008; Sekiyama and Unno 2023; Swamy et al. 2016; Terauchi 2010; Unno and Kobayashi 2009; Unno et al. 2018; Vazou et al. 2014; Vekris et al. 2016; Xi and Pfenning 1999; Zhu and Jagannathan 2013]. Such type systems allow the user to express a precise specification for a program as a type embedding logic formulas and their type checking (sometimes even type inference) (semi-)algorithms (semi-)automatically check whether the program conforms to the specification. While a variety of refinement type systems have been proposed for various classes of programming languages and features, including functional languages [Freeman and Pfenning 1991; Rondon et al. 2008; Vazou et al. 2014], object-oriented languages [Vekris et al. 2016], and delimited control operators [Sekiyama and Unno 2023], there has not been a satisfactory refinement type system for programming languages with algebraic effects and handlers.

In this work, we propose a new refinement type system for algebraic effects and handlers. A challenge with the precise verification in the presence of algebraic effects and handlers is the presence of the *delimited continuations*: they are the key ingredient of algebraic effects and handlers to implement a variety of computational effects, but they also complicate programs' control flow and make it difficult to statically discern what effects occur in what order. To address this challenge, we propose a novel concept that we call *answer refinement modification* (ARM for short), inspired by *answer type modification* (ATM) employed in certain type systems for delimited control operators such as `shift` and `reset` [Asai 2009; Danvy and Filinski 1990]. Similarly to ATM that can statically track how the use of delimited control operators influence the types of expressions, ARM can statically track how the use of algebraic effect operations (and the execution of the corresponding handlers) influence the refinements in the types of expressions, where the latter, as in prior refinement type systems, is used to precisely describe the *values*, rather than just their ordinary (i.e., non-refinement) types, computed by the expressions. Thus, our novel refinement type system supporting ARM can be used to precisely reason about programs with algebraic effects and handlers.

ATM and ARM are closely related: in fact, our refinement type system supports ATM, that is, our system allows the whole types and not just the refinements in them to be modified. As far as we know, the only prior (ordinary or refinement) type system for algebraic effects and handlers that supports ATM or ARM is a recent system of Cong and Asai [2022]. However, their system does not support refinement types (and so, obviously, no ARM), and moreover, even when compared as mechanisms for ordinary type systems, their ATM is less expressive than ours. We refer to Section 6 for detailed comparison.

While our system supports the full ATM, from the perspective of program verification, ARM alone, that is, allowing only modification in type refinements, is useful. Indeed, as in other refinement-type-based approaches, our aim is verification of programs typed in *ordinary* background type systems (such as the type systems of OCaml 5 and Koka [Leijen 2014] that do not support ATM), not to make more programs typable by extending the background type systems with ATM. As a proof of concept, we have extended the refinement type system and implemented a corresponding type checking and inference algorithm for a subset of OCaml 5 which comes with a built-in support for effect handlers, and evaluated it on a number of benchmark programs that use algebraic effects and handlers. The evaluation demonstrates that ARM is conceptually simple and practically useful.

Finally, a natural alternative to directly reasoning about a program with delimited continuations is to apply a continuation passing style (CPS) transformation that transforms the program to a pure program without delimited continuations. We investigate this alternative in the paper, and show that the approach is indeed possible by proposing a novel CPS transformation for algebraic effects and handlers that enjoys bidirectional (refinement-)type-preservation. Bidirectional type-preservation means that an expression is well-typed in the source language if and only if its CPS-transformed result is well-typed in the target language. This implies that we can use existing refinement type systems without support for effect handlers to verify programs with effect handlers by applying our CPS transformation. However, like other CPS transformations [Appel 1992; Cong and Asai 2018; Danvy and Filinski 1990; Hillerström et al. 2017; Plotkin 1975], ours makes global changes to the program and can radically change its structure, making it difficult for the programmer to recast the type checking and inference results back to the original program. Also, the CPS transformation is type directed and requires the program to be annotated by types conforming to our new type system, albeit only needing type "structures" without concrete refinement predicates. Moreover, in some cases, CPS-transformed expressions need extra parameters or higher-order predicate polymorphism to be typed as precisely as the source expressions, because the CPS transformation introduces higher-order continuation arguments. Nonetheless, our CPS transformation is novel, and we foresee that it would provide new interesting insights, as CPS transformations often do [Danvy and Filinski 1990], and be a useful tool for future studies on refinement type systems and effect handlers.

Our main contributions are summarized as follows.

- We show a sound refinement type system for algebraic effects and handlers, where ARM plays an important role.
- We have implemented the refinement type system for a subset of OCaml language with effect handlers, and evaluate it on a number of programs that use effect handlers.
- We define a bidirectionally-type-preserving CPS transformation which can be used to verify programs with effect handlers, and discuss pros and cons between direct type checking using our system and indirect type checking via the CPS transformation.

The rest of the paper is organized as follows. In Section 2, we briefly explain algebraic effects and handlers and ATM, and then describe the motivation for ARM and our system. Section 3 presents our language. We define its syntax, semantics and type system, present some typing examples, and show type safety of the language. Section 4 explains the implementation of the system. In Section 5, we provide the CPS transformation and discuss pros and cons between the direct type checking via our type system and the indirect type checking via CPS transformation. Finally, we describe related works in Section 6 and conclude the paper in Section 7.

2 OVERVIEW

We briefly overview algebraic effects and handlers, ATM, and ARM.

2.1 Algebraic Effects and Handlers

Algebraic effects and handlers enable users to define their own effects in a modular way. The modularity stems from separating the use of effects from their implementations: effects are performed via *operations* and implemented via *effect handlers* (or handlers for short). For example, consider the following program where $h_\mathrm{d} \triangleq \{x_r \mapsto x_r,\ \mathrm{decide}(x, k) \mapsto \mathrm{max}\ (k\ \mathbf{true})\ (k\ \mathbf{false})\}$:

$$\mathbf{with}\ h_\mathrm{d}\ \mathbf{handle}\ \mathbf{let}\ a = \mathbf{if}\ \mathrm{decide}\ ()\ \mathbf{then}\ 10\ \mathbf{else}\ 20\ \mathbf{in}\ \mathbf{let}\ b = \mathbf{if}\ \mathrm{decide}\ ()\ \mathbf{then}\ 1\ \mathbf{else}\ 2\ \mathbf{in}\ a - b$$

It calls an operation decide, which takes the unit value () and returns a Boolean value, to choose one of two integer values and then calculates the difference between the chosen values. Because operation calls invoke effects in algebraic effects, the operations work as interfaces of the effects.

An implementation of an effect is given by an effect handler. The program installs the handler h_d for decide using the handling construct. In general, a handling construct takes the form **with** h **handle** e, which means that a handler h defines interpretations of operations performed during the evaluation of the expression e; we call the expression e a *handled expression.* A handler consists of a single *return clause* and zero or more *operation clauses.* A return clause takes the form $x_r \mapsto e_r$, which determines the value of the handling construct by evaluating expression e_r with variable x_r that denotes the value of the handled expression. In the example, because the return clause is $x_r \mapsto x_r$, the handling construct simply returns the value of the handled expression. An operation clause takes the form $\mathrm{op}(x, k) \mapsto e$. It defines the interpretation of the operation op to be expression e with variable x that denotes the arguments to the operation. When the handled expression calls the operation op, the remaining computation up to the handling construct is suspended and instead the body e of the operation clause evaluates. Therefore, effect handlers behave like exception handlers by regarding operation calls as raising exceptions. However, effect handlers are equipped with the additional ability to resume the suspended computation. The suspended remaining computation, called a *delimited continuation*, is functionalized, and the body e of the operation clause can refer to it via the variable k.

Let us take a closer look at the behavior of the above example. Because the handled expression starts with the call to decide, the operation clause for decide given by h_d evaluates. The delimited continuation K of the first call to decide is

$$\mathbf{with}\ h_\mathrm{d}\ \mathbf{handle}\ (\mathbf{let}\ a = \mathbf{if}\ [\,]\ \mathbf{then}\ 10\ \mathbf{else}\ 20\ \mathbf{in}\ \ \mathbf{let}\ b = \mathbf{if}\ \mathrm{decide}\ ()\ \mathbf{then}\ 1\ \mathbf{else}\ 2\ \mathbf{in}\ \ a - b)$$

where $[\,]$ denotes the hole of the continuation. We write $K[e]$ for the expression obtained by filling the hole in K with expression e. Then, the functional representation of the delimited continuation K takes the form $\lambda y.K[y]$, and it is substituted for k in the body of the operation clause. Namely, the handling construct evaluates to $\mathrm{max}\ (v\ \mathbf{true})\ (v\ \mathbf{false})$ where $v = \lambda y.K[y]$. Note that the variable x of the operation clause for decide is replaced by the unit value (), but it is not referenced. The first argument v **true** to max reduces to $K[\mathbf{true}]$, that is,

$$\mathbf{with}\ h_\mathrm{d}\ \mathbf{handle}\ (\mathbf{let}\ a = \mathbf{if}\ \boxed{\mathbf{true}}\ \mathbf{then}\ 10\ \mathbf{else}\ 20\ \mathbf{in}\ \mathbf{let}\ b = \mathbf{if}\ \mathrm{decide}\ ()\ \mathbf{then}\ 1\ \mathbf{else}\ 2\ \mathbf{in}\ \ a - b)$$

(the grayed part represents the value by which the hole in K is replaced). Therefore, the expression v **true** evaluates to $\mathbf{with}\ h_\mathrm{d}\ \mathbf{handle}\ (\mathbf{let}\ b = \mathbf{if}\ \mathrm{decide}\ ()\ \mathbf{then}\ 1\ \mathbf{else}\ 2\ \mathbf{in}\ 10 - b)$. Again, decide is called and the continuation $K' \triangleq \mathbf{with}\ h_\mathrm{d}\ \mathbf{handle}\ (\mathbf{let}\ b = \mathbf{if}\ [\,]\ \mathbf{then}\ 1\ \mathbf{else}\ 2\ \mathbf{in}\ 10 - b)$ is captured. Then, the operation clause for decide evaluates after substituting $\lambda y.K'[y]$ for k. The expression $(\lambda y.K'[y])\ \mathbf{true}$ evaluates to $K'[\mathbf{true}]$, that is, $\mathbf{with}\ h_\mathrm{d}\ \mathbf{handle}\ (\mathbf{let}\ b = \mathbf{if}\ \boxed{\mathbf{true}}\ \mathbf{then}\ 1\ \mathbf{else}\ 2\ \mathbf{in}\ 10 - b)$ and then to $\mathbf{with}\ h_\mathrm{d}\ \mathbf{handle}\ 9$. Here, the handled expression is a value. Therefore, the return clause in the handler evaluates after substituting the value 9 for variable x_r. Because the return clause in h_d just returns x_r, the evaluation of $(\lambda y.K'[y])\ \mathbf{true}$ results in 9. Similarly, $(\lambda y.K'[y])\ \mathbf{false}$ evaluates to 8 (which is the result of binding b to 2). Therefore, $\mathrm{max}\ ((\lambda y.K'[y])\ \mathbf{true})\ ((\lambda y.K'[y])\ \mathbf{false})$

evaluates to max 9 8 and then to 9. In a similar way, v **false** calculates max $(20-1)$ $(20-2)$, that is, evaluates to 19, because a is bound to 20 and b is bound to each of 1 and 2 depending on the result of the second invocation of decide. Finally, the entire program evaluates to 19, which is the result of max (v **true**) (v **false**), that is, max 9 19.

2.2 Answer Type Modification and Answer Refinement Modification

An *answer type* is the type of the closest enclosing delimiter, or the return type of a delimited continuation. In the setting of algebraic effects and handlers, delimiters are handling constructs. For example, consider the following expression:

$$\textbf{let } x = \textbf{with } \{x_r \mapsto x_r,\ \mathrm{op}((), k) \mapsto k\ 0 < k\ 1\} \textbf{ handle } 1 + \mathrm{op}\ ()\ \textbf{in } c\,.$$

The delimited continuation of op () is $K'' \triangleq \textbf{with } \{x_r \mapsto x_r,\ \mathrm{op}((), k) \mapsto k\ 0 < k\ 1\} \textbf{ handle } 1 + [\ ]$. At first glance, the answer type of op () seems to be the integer type int since the handled computation in the continuation returns the integer $1 + n$ for an integer n given to fill the hole, and the return clause returns given values as they are. In other words, from the perspective of op (), the handling construct seems to give an integer value to the outer context **let** $x = [\ ]$ **in** c. However, after the operation call, the entire expression evaluates to **let** $x = v''\ 0 < v''\ 1$ **in** c where $v'' \triangleq \lambda y.K''[y]$. Now the handling construct becomes the expression $v''\ 0 < v''\ 1$, which gives a Boolean value to the outer context. That is, the answer type changes to the Boolean type bool. *Answer type modification* (ATM) is a mechanism to track this dynamic change on answer types.

ATM is not supported in existing type systems for effect handlers [Bauer and Pretnar 2013, 2015; Brady 2013; Kammar et al. 2013; Leijen 2017; Lindley et al. 2017; Plotkin and Pretnar 2013], with the exception of the one recently proposed by Cong and Asai [2022] (see Section 6 for comparison with their work). Such type systems require the answer types before and after an operation call to be unified (and so the example above will be rejected as ill-typed). Nonetheless, useful programming with effect handlers is still possible without ATM (which is why they are implemented in popular languages like OCaml without ATM).[1] For instance, the program in Section 2.1 is well-typed in existing (non-refinement) type systems for algebraic effects and handlers without ATM, since the return type of the continuation k in the decide clause (i.e., the answer type before the execution) is int and the return type of the decide clause (i.e., the answer type after the execution) is also int.

However, even if answer types are not modified, *actual values returned by delimited continuations usually change.* Let us see the program in Section 2.1 again. Focus on the first call to decide. When this is called, the operation clause receives the continuation $v = \lambda y.K[y]$, which returns 9 if applied to **true** and returns 19 if applied to **false**, as described previously. Therefore, v can be assigned the refinement type $(y : \mathrm{bool}) \to \{z : \mathrm{int} \mid z = (y\ ?\ 9 : 19)\}$, and thus the precise answer type before the execution is $\{z : \mathrm{int} \mid z = (y\ ?\ 9 : 19)\}$ where y is the Boolean value passed to the continuation. On the other hand, the clause for decide returns integer 19. Thus, the precise answer type after the operation call is $\{z : \mathrm{int} \mid z = 19\}$. Now the refinement in the answer type becomes different before and after the operation call. The same phenomenon happens in the second call to decide. When the second call evaluates, the handler receives the continuation $\lambda y.K'[y]$. It returns $a - 1$ if applied to **true** and returns $a - 2$ if applied to **false** (where a is either 10 or 20 depending on the result of the first call to decide). Thus, the answer type before the execution is $\{z : \mathrm{int} \mid z = (y\ ?\ a-1 : a-2)\}$. In contrast, the return value of the clause for decide is max $(a-1)$ $(a-2) = a - 1$, so the answer type after the execution is $\{z : \mathrm{int} \mid z = a - 1\}$. Here again, the refinement in the answer type changed by the operation call. We call this change *answer refinement modification* (ARM). Armed with ARM

[1] One could also argue that the absence of ATM is natural for algebraic effects and handlers because they are designed after concepts from universal algebra [Bauer 2018; Plotkin and Power 2001], and there, (algebraic) operations are usually expected to preserve types.

(pun intened), the refinement type system that we propose in this paper is able to assign the precise refinement type $\{z : \mathrm{int} \mid z = 19\}$ to the program, and more generally, the type $\{z : \mathrm{int} \mid z = v - x\}$ when the constants 10, 20, 1, and 2 are replaced by variables u, v, x, and y respectively with the assumption $u \le v \land x \le y$ (such an assumption on free variables can be given by refinement types in the top-level type environment). The example demonstrates that ARM is useful for precisely reasoning about programs with algebraic effects and handlers in refinement type systems. Indeed, without ARM, the most precise refinement type that a type system could assign to the example would be $\{z : \mathrm{int} \mid z \in \{8, 9, 18, 19\}\}$.

As another illuminating example, we show that ARM provides a new approach to the classic *strong update* problem [Foster et al. 2002]. It is well known that algebraic effects and handlers can implement mutable references by operations set and get, that respectively destructively updates and reads a mutable reference, and a handler that implements the operations by state-passing (see, e.g., [Pretnar 2015]). On programs with such a standard implementation of mutable references by algebraic effects and handlers, our refinement type system is able to reason flow-sensitively and derive refinement types that cannot be obtained with ordinary flow-insensitive reasoning. For instance, consider the following program where $h \triangleq \{x_r \mapsto \lambda s.x_r,\ \mathrm{set}(x, k) \mapsto \lambda s.k\ ()\ x,\ \mathrm{get}(x, k) \mapsto \lambda s.k\ s\ s\}$:

$$(\mathbf{with}\ h\ \mathbf{handle}\ \ (\mathrm{set}\ 3;\ \mathbf{let}\ n = \mathrm{get}\ ()\ \mathbf{in}\ \mathrm{set}\ 5;\ \mathbf{let}\ m = \mathrm{get}\ ()\ \mathbf{in}\ n + m))\ 0$$

Thanks to ARM, our type system can give the program the most precise type $\{z : \mathrm{int} \mid z = 8\}$, which would not be possible in a type system without ARM as it would conflate the two calls to set and fail to reason that the first get () returns 3 whereas the second get () returns 5. Roughly, ARM accomplishes the flow-sensitive reasoning about the changes in the state by tracking changes in the refinements in the answer types, albeit in a *backward* fashion as shown in Section 3.3.

Using this ability of ARM, we can also verify that effectful operations are used in a specific order. For example, consider operations open, close, read, and write for file manipulation being implemented using effect handlers. The use of these operations should conform to the regular scheme (open (read | write)* close)*. Our refinement type system can check if a program meets this requirement. For instance, consider the following recursive function:

$$\lambda x.\ \mathbf{while}\ (\star)\ \{\mathrm{open}\ x;\ \mathbf{while}\ (\star)\ \{\mathbf{let}\ y = \mathrm{read}()\ \mathbf{in}\ \mathrm{write}\ (y\texttt{\^{}"X"})\};\ \mathrm{close}\ ()\}$$

where **while** $(\star)$ $\{c\}$ loops computation c and terminates nondeterministically, and the binary operation (^) concatenates given strings (operation read is supposed to return a string).[2] The function repeats opening the specified file x and closing it after reading from and writing to the file zero or more times. Thus, this function follows the discipline of the file manipulation operations. We will show in Section 3.3 how ARM enables us to check it formally and detect the invalid use of the operations if any.

3 LANGUAGE

This section presents our language with algebraic effects and handlers. The semantics is formalized using evaluation contexts like in Leijen [2017], and the type system is a novel refinement type system with ARM (and ATM).

3.1 Syntax and Semantics

The upper half of Figure 1 shows the syntax of our language. It indicates that expressions are split into values, ranged over by v, and computations, ranged over by c, as in the fine-grain call-by-value style of Levy et al. [2003]. Values, which are effect-free expressions in a canonical form, consist

[2] For simplicity, we assume that the clause of open creates an object for a specified file and stores it in a reference implemented by an effect handler, and the clauses of the other operations refer to the stored object to manipulate the file.

Syntax

$$p ::= \textbf{true} \mid \textbf{false} \mid \cdots \quad v ::= x \mid p \mid \textbf{rec}(f,x).c \quad K ::= [\,] \mid \textbf{let}\ x = K\ \textbf{in}\ c$$
$$c ::= \textbf{return}\ v \mid \mathrm{op}\ v \mid v_1\ v_2 \mid \textbf{if}\ v\ \textbf{then}\ c_1\ \textbf{else}\ c_2 \mid \textbf{let}\ x = c_1\ \textbf{in}\ c_2 \mid \textbf{with}\ h\ \textbf{handle}\ c$$
$$h ::= \{\textbf{return}\ x_r \mapsto c_r, (\mathrm{op}_i(x_i,k_i) \mapsto c_i)_i\}$$

Evaluation rules $\boxed{c \longrightarrow c'}$

$$\frac{c_1 \longrightarrow c_1'}{\textbf{let}\ x = c_1\ \textbf{in}\ c_2 \longrightarrow \textbf{let}\ x = c_1'\ \textbf{in}\ c_2}\ (\text{E-Let}) \qquad \frac{}{\textbf{let}\ x = \textbf{return}\ v\ \textbf{in}\ c_2 \longrightarrow c_2[v/x]}\ (\text{E-LetRet})$$

$$\frac{}{\textbf{if true then}\ c_1\ \textbf{else}\ c_2 \longrightarrow c_1}\ (\text{E-IfT}) \qquad \frac{}{\textbf{if false then}\ c_1\ \textbf{else}\ c_2 \longrightarrow c_2}\ (\text{E-IfF})$$

$$\frac{}{(\textbf{rec}(f,x).c)\ v \longrightarrow c[v/x][(\textbf{rec}(f,x).c)/f]}\ (\text{E-App}) \qquad \frac{}{p\ v \longrightarrow \zeta(p,v)}\ (\text{E-Prim})$$

below, let $h = \{\textbf{return}\ x_r \mapsto c_r, (\mathrm{op}_i(x_i,k_i) \mapsto c_i)_i\}$

$$\frac{c \longrightarrow c'}{\textbf{with}\ h\ \textbf{handle}\ c \longrightarrow \textbf{with}\ h\ \textbf{handle}\ c'}\ (\text{E-Hndl})$$

$$\frac{}{\textbf{with}\ h\ \textbf{handle return}\ v \longrightarrow c_r[v/x_r]}\ (\text{E-HndlRet})$$

$$\frac{}{\textbf{with}\ h\ \textbf{handle}\ K[\mathrm{op}_i\ v] \longrightarrow c_i[v/x_i][(\lambda y.\textbf{with}\ h\ \textbf{handle}\ K[\textbf{return}\ y])/k_i]}\ (\text{E-HndlOp})$$

Fig. 1. Syntax and evaluation rules.

of variables x, primitive values p, and (recursive) functions $\textbf{rec}(f,x).c$ where variable f denotes the function itself for recursive calls in the body c. If f does not occur in c, we simply write $\lambda x.c$. Computations, which are possibly effectful expressions, consist of six kinds of constructs. A value-return **return** v lifts a value v to a computation. An operation call op v performs the operation op with the argument v. A function application $v_1\ v_2$, conditional branch **if** v **then** c_1 **else** c_2, and let-expression **let** $x = c_1$ **in** c_2 are standard. Note that functions, arguments, and conditional expressions are restricted to values, but this does not reduce expressivity because, e.g., a conditional branch **if** c **then** c_1 **else** c_2 can be expressed as **let** $x = c$ **in if** x **then** c_1 **else** c_2 using a fresh variable x. A handling construct **with** h **handle** c handles operations performed during the evaluation of the handled computation c using the clauses in the handler h. A handler $\{\textbf{return}\ x_r \mapsto c_r, (\mathrm{op}_i(x_i,k_i) \mapsto c_i)_i\}$ has a return clause $\textbf{return}\ x_r \mapsto c_r$ where the variable x_r denotes the value of the handled computation c, and an operation clause $\mathrm{op}_i(x_i,k_i) \mapsto c_i$ for each operation op_i where the variables x_i and k_i denote the argument to op_i and the continuation from the invocation of op_i, respectively. The notions of free variables and substitution are defined as usual. We write $c[v/x]$ for the computation obtained by substituting the value v for the variable x in the computation c. We use similar notation to substitute values for variables in types and substitute types for type variables.

The semantics of the language is defined by the evaluation relation $\longrightarrow$, which is the smallest binary relation over computations satisfying the evaluation rules in the lower half of Figure 1. The evaluation of a let-expression **let** $x = c_1$ **in** c_2 begins by evaluating the computation c_1. When c_1 returns a value, the computation c_2 evaluates after substituting the return value for x. The evaluation rules for conditional branching and function application are standard. The result of applying a primitive value relies on the metafunction ζ, which maps pairs of a primitive value and an argument value to computations. For a handling construct **with** h **handle** c, the handled computation c evaluates first. When c returns a value, the body of the return clause in the handler h evaluates with the return value. If the evaluation of c encounters an operation call $\mathrm{op}_i\ v$, its delimited continuation, which is represented as a pure evaluation context K defined in Figure 1, is captured. Then, the body c_i of the operation clause $\mathrm{op}_i(x_i, k_i) \mapsto c_i$ for op_i in the handler h evaluates after substituting the argument v and the function $\lambda y.\mathbf{with}\ h\ \mathbf{handle}\ K[\mathbf{return}\ y]$ for variables x_i and k_i, respectively. Note that the function substituted for k_i wraps the delimited continuation $K[\mathbf{return}\ y]$ by the handling construct with the handler h. It means that the operation calls in $K[\mathbf{return}\ y]$ are handled by the handler h. Our semantics assumes that the handler h provides operation clauses for all the operations performed by the handled computation c. Our type system ensures that this assumption holds on any well-typed computations. However, our language can also implement the forwarding semantics by encoding: given a handler that does not contain an operation clause for op, we add to the handler an operation clause $\mathrm{op}(x, k) \mapsto \mathbf{let}\ y = \mathrm{op}\ x\ \mathbf{in}\ k\ y$.[3]

3.2 Type System

$$
\begin{array}{rrcl@{\qquad}rrcl}
\text{term} & t & ::= & x \mid \ldots & \text{formula} & \phi & ::= & A(\widetilde{t}) \mid \ldots \\
\text{predicate} & A & ::= & X \mid \ldots & \text{base type} & B & ::= & \mathrm{bool} \mid \ldots
\end{array}
$$

$$
\begin{array}{rrcl}
\text{value type} & T & ::= & \{x : B \mid \phi\} \mid (x : T) \to C \\
\text{computation type} & C & ::= & \Sigma \triangleright T \ /\ S \\
\text{operation signature} & \Sigma & ::= & \{\widetilde{(\mathrm{op}_i : \forall X_i : \widetilde{B}_i.F_i)_i}\} \\
 & F & ::= & (x : T_1) \to ((y : T_2) \to C_1) \to C_2 \\
\text{control effect} & S & ::= & \square \mid (\forall x.C_1) \Rightarrow C_2 \\
\text{typing context} & \Gamma & ::= & \emptyset \mid \Gamma, x : T \mid \Gamma, X : \widetilde{B}
\end{array}
$$

Fig. 2. Type syntax.

Figure 2 shows the syntax of types. As in prior refinement type systems [Bengtson et al. 2011; Rondon et al. 2008; Unno and Kobayashi 2009], our type system allows a type specification for values of base types, ranged over by B, such as bool and int, to be refined using logic formulas, ranged over by ϕ. Our type system is parameterized over a logic. We assume that the logic is a predicate logic where: terms, denoted by t, include variables x; predicates, denoted by A, include predicate variables X; and each primitive value p can be represented as a term. Throughout the paper, we use the over-tilde notation to denote a sequence of entities. For example, $\widetilde{t}$ represents a sequence $t_1, \cdots, t_n$ of some terms $t_1, \ldots, t_n$, and then $A(\widetilde{t})$ represents a formula $A(t_1, \cdots, t_n)$. We also assume that base types include at least the Boolean type bool.

Types consist of value and computation types, which are assigned to values and computations, respectively. A value type, denoted by T, is either a refinement type $\{x : B \mid \phi\}$, which is assigned to a value v of base type B such that the formula $\phi[v/x]$ is true, or a dependent function type $(x : T) \to C$, which is assigned to a function that, given an argument v of the type T, performs the computation specified by the type $C[v/x]$. We abbreviate $(x : T) \to C$ as $T \to C$ if x does not occur in C, and $\{z : B \mid \mathbf{true}\}$ as B.

A computation type is formed by three components: an operation signature, which specifies operations that a computation may perform; a value type, which specifies the value that the computation returns if any; and a control effect, which specifies how the computation modifies the answer type via operation call.

Control effects, denoted by S, are inspired by the formalism of Sekiyama and Unno [2023] who extended control effects in simple typing [Materzok and Biernacki 2011] to dependent typing. A control effect is either pure or impure. The pure control effect $\square$ means that a computation calls

[3]We employ the semantics without forwarding in the body of the paper to simplify the typing rule for handling constructs. The supplementary material shows an extended typing rule for handling constructs that natively supports forwarding.

no operation. An impure control effect is given to a computation that may perform operations, specifying how the execution of the computation modifies its answer type. Impure control effects take the form $(\forall x.C_1) \Rightarrow C_2$ where variable x is bound in computation type C_1. We write $C_1 \Rightarrow C_2$ when x does not occur in C_1. In what follows, we first illustrate impure control effects in the simple, nondependent form $C_1 \Rightarrow C_2$ and then extend to the fully dependent form $(\forall x.C_1) \Rightarrow C_2$ that can specify the behavior of captured continuations using the input (denoted by x) to the continuations.

A control effect $C_1 \Rightarrow C_2$ represents the answer type of a program changes from type C_1 to type C_2. When it is assigned to a computation c, the initial answer type C_1 specifies how the continuation of the computation c up to the closest handing construct behaves, and the final answer type C_2 specifies what can be guaranteed for the *meta-context*, i.e., the context of the closest handling construct. To see the idea more concretely, revisit the first example in Section 2.2:

$$\textbf{let } x = \textbf{with } \{x_r \mapsto x_r,\ \mathsf{op}((), k) \mapsto k\ 0 < k\ 1\} \textbf{ handle } 1 + \mathsf{op}\ () \textbf{ in } c\ .$$

Let h be the handler in the example. Focusing on the operation call op (), we can find that it captures the continuation **with** h **handle** $1 + [\ ]$. Because the continuation behaves as if it is a pure function returning integers, the initial answer type of op () turns out to be the computation type int / □ (we omit Σ for a while; it will be explained shortly). Furthermore, by the operation call, the handling construct **with** h **handle** $1 + \mathsf{op}\ ()$ is replaced with the body $k\ 0 < k\ 1$ of op's clause in h and the functional representation v of the continuation is substituted for k. It means that the meta-context **let** $x = [\ ]$ **in** c of the operation call takes the computation $v\ 0 < v\ 1$, which is of type bool / □ (note that $v\ 0 < v\ 1$ is pure because v is a pure function). Therefore, the final answer type of op () is bool / □. As a result, the impure control effect of op () is int / □ ⇒ bool / □.

Sekiyama and Unno [2023] extended the simple form of impure control effects to a dependent form $(\forall x.C_1) \Rightarrow C_2$, where the initial answer type C_1 can depend on inputs, denoted by variable x, to continuations. For instance, consider the continuation **with** h **handle** $1+[\ ]$ captured in the above example. When passed an integer n, it returns $1 + n$. Using the dependent form of impure control effects, we can describe such behavior by the control effect $(\forall x.\{y : \mathsf{int} \mid y = x{+}1\} / \square) \Rightarrow \mathsf{bool} / \square$, where x represents the input to the continuation and the refinement type $\{y : \mathsf{int} \mid y = x + 1\}$ precisely specifies the return value of the continuation for input x. The type of x is matched with the continuation's input type. Since the continuation of op () takes integers, the type assigned to x is int. In general, given a computation type $T\ /\ (\forall x.C_1) \Rightarrow C_2$, the type T is assigned to the variable x because it corresponds to the input type of the continuations of computations given that computation type. The type information refined by dependent impure control effects is exploited in typechecking operation clauses. In the example, our type system typechecks the body of op's clause by assigning the function type $(x : \mathsf{int}) \rightarrow \{y : \mathsf{int} \mid y = x + 1\} / \square$ to the continuation variable k. Then, since the body is $k\ 0 < k\ 1$, its type—i.e., the final answer type—can be refined to $\{z : \mathsf{bool} \mid z = \textbf{true}\} / \square$. Hence, the type system can assign control effect $(\forall x.\{y : \mathsf{int} \mid y = x + 1\} / \square) \Rightarrow \{z : \mathsf{bool} \mid z = \textbf{true}\} / \square$ to the operation call and ensure that the meta-context takes **true** finally (if the handling construct terminates). We will demonstrate the expressivity and usefulness of dependent control effects in more detail in Section 3.3.

Operation signatures, denoted by Σ, are sets of pairs of an operation name and a type scheme. We write $\widetilde{(\cdot)_i}$ to denote a sequence of entities indexed by i. The type scheme associated with an operation op is in the form $\forall X : \widetilde{B}.(x : T_1) \rightarrow ((y : T_2) \rightarrow C_1) \rightarrow C_2$, where the types T_1 and T_2 are the input and output types, respectively, of the operation and the types C_1 and C_2 are the initial and final answer types, respectively, of the operation call for op. Recall that the initial answer type C_1 corresponds to the return type of delimited continuations captured by the call to op, and that the continuations take the return values of the operation call. Therefore, the function type

$(y : T_2) \rightarrow C_1$ represents the type of the captured delimited continuations. Note that the variable y denotes values passed to the continuations. Furthermore, the final answer type C_2 corresponds to the type of the operation clause for op in the closest enclosing handler. Therefore, the operation clause $\mathrm{op}(x, k) \mapsto c$ in the handler is typed by checking that the body c is of the type C_2 with the assumption that argument variable x is of the type T_1 and the continuation variable k is of the type $(y : T_2) \rightarrow C_1$. A notable point of the type scheme is that it can be parameterized over predicates. The predicate variables $\widetilde{X}$ abstract over the predicates, and the annotations $\widetilde{B}$ represent the (base) types of the arguments to the predicates. This allows calls to the same operation in different contexts to have different control effects, which is crucial for precisely verifying programs with algebraic effects and handlers as we will show in Section 3.3. It is also noteworthy that operation signatures include not only operation names but also type schemes as in Kammar et al. [2013] and Kammar and Pretnar [2017]. It allows an operation to have different types depending on the contexts where it is used. Another approach is to include only operation names and assumes that unique types are assigned to them globally as in, e.g., Bauer and Pretnar [2013] and Leijen [2017]. We decided to assign types to operations locally because it makes the type system more flexible in that the types of operations can be refined depending on contexts if needed.

Typing contexts Γ are lists of variable bindings $x : T$ and predicate variable bindings $X : \widetilde{B}$. We write Γ, ϕ for $\Gamma, x : \{z : B \mid \phi\}$ where x and z are fresh. The notions of free variables, free predicate variables, and predicate substitution are defined as usual.

Well-formedness of typing contexts, value types, and computation types, whose judgments are in the forms $\vdash \Gamma$, $\Gamma \vdash T$, and $\Gamma \vdash C$, respectively, are defined straightforwardly by following Sekiyama and Unno [2023]. We refer to the supplementary material for detail.

Typing judgements for values and computations are in the forms $\Gamma \vdash v : T$ and $\Gamma \vdash c : C$, respectively. Figure 3 shows the typing rules. By (T-CVar), a variable x of a refinement type is assigned a type which states that the value of this type is exactly x. For a variable of a non-refinement type (i.e., a function type in our language), the rule (T-Var) assigns the type associated with the variable in the typing context. The rule (T-Prim) uses the mapping ty to type primitive values p. We assume that ty assigns an appropriate value type to every primitive value. We refer to the supplementary material for the formalization of the assumption. The rule (T-Fun) for functions, (T-App) for function applications, and (T-If) for conditional branches are standard in refinement type systems (with support for value-dependent refinements). The rules (T-VSub) and (T-CSub) allow values and computations, respectively, to be typed at supertypes of their types. We will define subtyping shortly. By (T-Ret), a value-return **return** v has a computation type where the operation signature is empty, the return value type is the type of v, and the control effect is pure.

To type a let-expression **let** $x = c_1$ **in** c_2, either the rule (T-LetP) or (T-LetIp) is used. Both of them require that the types of the sub-expressions c_1 and c_2 have the same operation signature Σ and then assign Σ to the type of the entire let-expression. The typing context for c_2 is extended by $x : T_1$ with the value type T_1 of c_1, but x cannot occur in Σ and T_2 (as well as C_{21} in (T-LetIp)) to prevent the leakage of x from its scope. On the other hand, the two rules differ in how they treat control effects. When both of the control effects of c_1 and c_2 are pure, the rule (T-LetP) is used. It states that the control effect of the entire let-expression is also pure. When both are impure, the rule (T-LetIp) is used. It states that the control effect of the let-expression results in an impure control effect that is composed of the control effects of c_1 and c_2. Note that, even when one of the control effects of c_1 and c_2 is pure and the other is impure, we can view both of them as impure effects via subtyping because it allows converting a pure control effect to an impure control effect, as shown later. We first explain how the composition works in the non-dependent form. Let the control effect of c_1 be $C_{11} \Rightarrow C_{12}$ and that of c_2 be $C_{21} \Rightarrow C_{22}$, and assume that a control effect

Typing rules for values $\boxed{\Gamma \vdash v : T}$

$$\frac{\vdash \Gamma \quad \Gamma(x) = \{z : B \mid \phi\}}{\Gamma \vdash x : \{z : B \mid z = x\}}\text{(T-CVar)} \quad \frac{\vdash \Gamma \quad \forall y, B, \phi. \Gamma(x) \neq \{y : B \mid \phi\}}{\Gamma \vdash x : \Gamma(x)}\text{(T-Var)} \quad \frac{\vdash \Gamma}{\Gamma \vdash p : ty(p)}\text{(T-Prim)}$$

$$\frac{\Gamma, f : (x : T) \to C, x : T \vdash c : C}{\Gamma \vdash \mathbf{rec}(f, x).c : (x : T) \to C}\text{(T-Fun)} \quad \frac{\Gamma \vdash v : T_1 \quad \Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2}{\Gamma \vdash v : T_2}\text{(T-VSub)}$$

Typing rules for computations $\boxed{\Gamma \vdash c : C}$

$$\frac{\Gamma \vdash v : T}{\Gamma \vdash \mathbf{return}\ v : \emptyset \triangleright T \ / \ \square}\text{(T-Ret)} \quad \frac{\Gamma \vdash v_1 : (x : T) \to C \quad \Gamma \vdash v_2 : T}{\Gamma \vdash v_1\ v_2 : C[v_2/x]}\text{(T-App)}$$

$$\frac{\begin{array}{c}\Gamma \vdash v : \{x : \mathrm{bool} \mid \phi\} \\ \Gamma, v = \mathbf{true} \vdash c_1 : C \quad \Gamma, v = \mathbf{false} \vdash c_2 : C\end{array}}{\Gamma \vdash \mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : C}\text{(T-If)} \quad \frac{\Gamma \vdash c : C_1 \quad \Gamma \vdash C_1 <: C_2 \quad \Gamma \vdash C_2}{\Gamma \vdash c : C_2}\text{(T-CSub)}$$

$$\frac{\begin{array}{c}\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ / \ \square \\ \Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \ / \ \square \\ x \notin fv(T_2) \cup fv(\Sigma)\end{array}}{\Gamma \vdash \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 : \Sigma \triangleright T_2 \ / \ \square}\text{(T-LetP)} \quad \frac{\begin{array}{c}\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ / \ (\forall x.C) \Rightarrow C_{12} \\ \Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \ / \ (\forall y.C_{21}) \Rightarrow C \\ x \notin fv(T_2) \cup fv(\Sigma) \cup (fv(C_{21}) \setminus \{y\})\end{array}}{\Gamma \vdash \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 : \Sigma \triangleright T_2 \ / \ (\forall y.C_{21}) \Rightarrow C_{12}}\text{(T-LetIp)}$$

$$\frac{\Sigma \ni \mathsf{op} : \forall \widetilde{X : \widetilde{B}}.(x : T_1) \to ((y : T_2) \to C_1) \to C_2 \quad \Gamma \vdash \Sigma \quad \widetilde{\Gamma \vdash A : \widetilde{B}} \quad \Gamma \vdash v : T_1[\widetilde{A/X}]}{\Gamma \vdash \mathsf{op}\ v : \Sigma \triangleright T_2[\widetilde{A/X}][v/x] \ / \ ((\forall y.C_1) \Rightarrow C_2)[\widetilde{A/X}][v/x]}\text{(T-Op)}$$

$$\frac{\begin{array}{c}h = \{\mathbf{return}\ x_r \mapsto c_r, (\mathsf{op}_i(x_i, k_i) \mapsto c_i)_i\} \quad \Gamma \vdash c : \Sigma \triangleright T \ / \ (\forall x_r.C_1) \Rightarrow C_2 \\ \Gamma, x_r : T \vdash c_r : C_1 \quad \left(\Gamma, \widetilde{X_i : \widetilde{B}_i}, x_i : T_{1i}, k_i : (y_i : T_{2i}) \to C_{1i} \vdash c_i : C_{2i}\right)_i \\ \Sigma = \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : T_{1i}) \to ((y_i : T_{2i}) \to C_{1i}) \to C_{2i})_i\}\end{array}}{\Gamma \vdash \mathbf{with}\ h\ \mathbf{handle}\ c : C_2}\text{(T-Hndl)}$$

Fig. 3. Typing rules.

$C_1 \Rightarrow C_2$ is assigned to the let-expression. First, recall that the type C_1 expresses the return type of the continuation of the let-expression up to the closest handling construct and that the closest handling construct is replaced by a computation of the type C_2. Based on this idea, the types C_1 and C_2 can be determined as follows. First, because the delimited continuation of the let-expression is matched with that of the computation c_2, the initial answer type C_{21} of c_2 expresses the return type of the delimited continuation of the let-expression. Therefore, the type C_1 should be matched with the type C_{21}. Second, because the closest handling construct enclosing the let-expression is the same as the one enclosing the sub-computation c_1, the type C_2 should be matched with the final answer type C_{12} of c_1. Therefore, the control effect $C_1 \Rightarrow C_2$ should be matched with $C_{21} \Rightarrow C_{12}$, as stated in (T-LetIp). Furthermore, the rule (T-LetIp) requires that the initial answer type C_{11} of c_1 to be the same as the final answer type C_{22} of c_2. This requirement is explained as follows. First, the computation c_1 expects its delimited continuation to behave as specified by the type C_{11}. The delimited continuation of c_1 first evaluates the succeeding computation c_2. The final answer type C_{22} of c_2 expresses that the closest handling construct enclosing c_2 behaves as specified by the type C_{22}. Because the closest handling construct enclosing c_2 corresponds to the top-level handling construct in the delimited continuation of c_1, the type C_{11} should be matched with the type C_{22}. We now extend to the fully dependent form. From the discussion thus far, we can let the control effects of c_1, c_2, and the let-expression be $(\forall x_1.C) \Rightarrow C_{12}$, $(\forall x_2.C_{21}) \Rightarrow C$, and $(\forall y.C_{21}) \Rightarrow C_{12}$ respectively, for some variables x_1, x_2, and y. Then, the constraints on the names of these variables are determined as follows. First, the input to the delimited continuation of c_1, which is denoted by the variable x_1, should be matched with the evaluation result of c_1. Then, since the let-expression binds the variable x to the evaluation result of c_1, the variable x_1 is matched with x. Second, because the delimited continuation of c_2 is matched with that of the let-expression, the inputs to them should be matched with each other. They are denoted by the variables x_2 and y respectively, and hence the variable x_2 is matched with y.

The rule (T-Hndl) for handling constructs **with** h **handle** c is one of the most important rules of our system. It assumes that the handled computation c is of a type $\Sigma \triangleright T \ / \ (\forall x_r.C_1) \Rightarrow C_2$, where the control effect is impure. Even when c is pure (i.e., performs no operation), it can have an impure control effect via subtyping. Because the type of the handling construct represents how the expression is viewed from the context, it should be matched with the final answer type C_2 of the handled computation c. The premises in the second line define typing disciplines that the clauses in the installed handler h have to satisfy. First, let us consider the return clause **return** $x_r \mapsto c_r$. Because the variable x_r denotes the return value of the handled computation c, the value type T of c is assigned to x_r. Moreover, since the return clause is executed after evaluating c, the body c_r is the delimited continuation of c. Therefore, the type of c_r should be matched with the initial answer type C_1 of c. Because the variable x_r bound in the return clause can be viewed as the input to the delimited continuation c_r, it should be matched with the variable x_r bound in the impure control effect $(\forall x_r.C_1) \Rightarrow C_2$. Operation clauses are typed using the corresponding type schemes in the operation signature Σ, as explained above. Note that the rule also requires the installed handler h to include a clause for each of the operations in Σ, i.e., those that c may perform.

The rule (T-Op) for operation calls is another important rule. Consider an operation call op v. The rule assumes that an enclosing handler addresses the operation op by requiring that an operation signature Σ assigned to the operation call include the operation op with a type scheme $\forall \widetilde{X : B}.(x : T_1) \rightarrow ((y : T_2) \rightarrow C_1) \rightarrow C_2$, and instantiates the predicate variables $\widetilde{X}$ in the type scheme with well-formed predicates $\widetilde{A}$ to reflect the contextual information of the operation call. Then, it checks that the argument v has the input type $T_1[\widetilde{A/X}]$ of the operation. Finally, the rule assigns the output type $T_2[\widetilde{A/X}][v/x]$ of the operation as the value type of the operation call, and $C_1[\widetilde{A/X}][v/x]$ and $C_2[\widetilde{A/X}][v/x]$ as the initial and final answer types of the operation call, respectively (note that the types T_2, C_1, and C_2 are parameterized over predicates and arguments).

The type system defines four kinds of subtyping judgments: $\Gamma \vdash T_1 <: T_2$ for value types, $\Gamma \vdash C_1 <: C_2$ for computations types, $\Gamma \vdash \Sigma_1 <: \Sigma_2$ for operation signatures, and $\Gamma \mid T \vdash S_1 <: S_2$ for control effects. Figure 4 shows the subtyping rules. The subtyping rules for control effects are adopted from the work of Sekiyama and Unno [2023], which extends subtyping for control effects given by Materzok and Biernacki [2011] to dependent typing. The rules (S-Rfn) and (S-Fun) for value types are standard. The judgement $\Gamma \models \phi$ in (S-Rfn) means the semantic validity of the formula ϕ under the assumption Γ. Subtyping between operation signatures is determined by (S-Sig). This rule is based on the observation that an operation signature Σ represents the types of operation clauses in handlers, as seen in (T-Hndl). Then, the rule (S-Sig) can be viewed as defining a subtyping relation between the types of handlers (except for return clauses): a handler for operations in Σ_1 can be used as one for operations in Σ_2 if every operation op in Σ_2 is included in Σ_1 (i.e., the handler has an operation clause for every op in Σ_2) and the type scheme of op in Σ_1

Subtyping rules $\boxed{\Gamma \vdash T_1 <: T_2}$ $\boxed{\Gamma \vdash \Sigma_1 <: \Sigma_2}$ $\boxed{\Gamma \vdash C_1 <: C_2}$ $\boxed{\Gamma \mid T \vdash S_1 <: S_2}$

$$\frac{\Gamma, x : B \models \phi_1 \implies \phi_2}{\Gamma \vdash \{x : B \mid \phi_1\} <: \{x : B \mid \phi_2\}}\text{(S-RFN)} \qquad \frac{\Gamma \vdash T_2 <: T_1 \quad \Gamma, x : T_2 \vdash C_1 <: C_2}{\Gamma \vdash (x : T_1) \to C_1 <: (x : T_2) \to C_2}\text{(S-FUN)}$$

$$\frac{(\Gamma, \widetilde{X_i : \widetilde{B}_i} \vdash F_{1i} <: F_{2i})_i}{\Gamma \vdash \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.F_{1i})_i, (\mathsf{op}'_i : \forall \widetilde{X'_i : \widetilde{B'}_i}.F'_i)_i\} <: \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.F_{2i})_i\}}\text{(S-SIG)}$$

$$\frac{\Gamma \vdash \Sigma_2 <: \Sigma_1 \quad \Gamma \vdash T_1 <: T_2 \quad \Gamma \mid T_1 \vdash S_1 <: S_2}{\Gamma \vdash \Sigma_1 \triangleright T_1 \ / \ S_1 <: \Sigma_2 \triangleright T_2 \ / \ S_2}\text{(S-COMP)} \qquad \frac{}{\Gamma \mid T \vdash \square <: \square}\text{(S-PURE)}$$

$$\frac{\Gamma, x : T \vdash C_{21} <: C_{11} \quad \Gamma \vdash C_{12} <: C_{22}}{\Gamma \mid T \vdash (\forall x.C_{11}) \Rightarrow C_{12} <: (\forall x.C_{21}) \Rightarrow C_{22}}\text{(S-ATM)} \qquad \frac{\Gamma, x : T \vdash C_1 <: C_2 \quad x \notin fv(C_2)}{\Gamma \mid T \vdash \square <: (\forall x.C_1) \Rightarrow C_2}\text{(S-EMBED)}$$

Fig. 4. Subtyping rules.

is a subtype of the type scheme of op in Σ_2 (i.e., the operation clause for op in the handler works as one for op in Σ_2). Given a computation type $C_1 \triangleq \Sigma_1 \triangleright T_1 \ / \ S_1$ and its supertype $C_2 \triangleq \Sigma_2 \triangleright T_2 \ / \ S_2$, a handler for operations performed by the computations of the type C_2 (i.e., the operations in Σ_2) is required to be able to handle operations performed by the computations of the type C_1 (i.e., the operations in Σ_1) because the subtyping allows deeming the computations of C_1 to be of C_2. The safety of such handling is ensured by requiring $\Sigma_2 <: \Sigma_1$. In the rule (S-COMP), the first premise represents this requirement. The second premise $\Gamma \vdash T_1 <: T_2$ in (S-COMP) allows viewing the return values of the computations of the type C_1 as those of the type C_2. The third premise $\Gamma \mid T_1 \vdash S_1 <: S_2$ expresses that the use of effects by the computations of the type C_1 is subsumed by the use of effects allowed by the type C_2. It is derived by the last three rules: (S-PURE), (S-ATM), and (S-EMBED). The rule (S-PURE) just states reflexivity of the pure control effect. If both S_1 and S_2 are impure, the rule (S-ATM) is applied. Because initial answer types represent the assumptions of computations on their contexts, (S-ATM) allows strengthening the assumptions by being contravariant in them. By contrast, because final answer types represent the guarantees of how enclosing handling constructs behave, (S-ATM) allows weakening the guarantees by being covariant in them. Note that the typing context for the initial answer types is extended with the binding $x : T_1$ because they may reference the inputs to the continuations via the variable x and the inputs are of the type T_1. Finally, the rule (S-EMBED) allows converting the pure control effect to an impure control effect $(\forall x.C_1) \Rightarrow C_2$. Because a computation c with the pure control effect performs no operation, what is guaranteed for the behavior of the handling construct enclosing c coincides with what is assumed on c's delimited continuation. Because the guarantee and assumption are specified by the types C_2 and C_1, respectively, if C_1 is matched with C_2—more generally, the "assumption" C_1 implies the "guarantee" C_2—the pure computation c can be viewed as the computation with the impure control effect $(\forall x.C_1) \Rightarrow C_2$. The first premise in (S-EMBED) formalizes this idea. Note that, because the variable x is bound in the type C_1, the rule (S-EMBED) disallows x to occur in the type C_2.

Finally, we state the type safety of our system. Its proof, via progress and subject reduction, is given in the supplementary material. We define $\longrightarrow^*$ as the reflexive, transitive closure of the one-step evaluation relation $\longrightarrow$.

THEOREM 3.1 (TYPE SAFETY). *If* $\emptyset \vdash c : \Sigma \triangleright T \ / \ S$ *and* $c \longrightarrow^* c'$, *then one of the following holds: (1)* $c' = \mathbf{return}\ v$ *for some* v *such that* $\emptyset \vdash v : T$; *(2)* $c' = K[\mathsf{op}\ v]$ *for some* K, op, *and* v *such that* $\mathsf{op} \in dom(\Sigma)$; *or (3)* $c' \longrightarrow c''$ *for some* c'' *such that* $\emptyset \vdash c'' : \Sigma \triangleright T \ / \ S$.

3.3 Examples

In this section, we demonstrate how our type system verifies programs with algebraic effects and handlers by showing typing derivations of a few examples. Here, we abbreviate a pure computation type $\{\} \triangleright T \;/\; \square$ to T and omit the empty typing context from typing and subtyping judgments. For simplicity, we often write $c_1\ c_2$ for an expression $\mathbf{let}\ x_1 = c_1\ \mathbf{in}\ \mathbf{let}\ x_2 = c_2\ \mathbf{in}\ x_1\ x_2$ where x_1 does not occur in c_2. Furthermore, we deal with a pure computation as if it is a value. For example, we write $\mathbf{return}\ c$ for a computation $\mathbf{let}\ x = c\ \mathbf{in}\ \mathbf{return}\ x$ if c is pure (e.g., as $\mathbf{return}\ a - b$).

3.3.1 Example 1: Nondeterministic Computation. We first revisit the example presented in Section 2.1. In our language, it can be expressed as follows:

$$\begin{array}{l}\mathbf{with}\ h\ \mathbf{handle}\ (\mathbf{let}\ a = (\mathbf{let}\ y = \mathrm{decide}\ ()\ \mathbf{in}\ \mathbf{if}\ y\ \mathbf{then}\ \mathbf{return}\ 10\ \mathbf{else}\ \mathbf{return}\ 20)\ \mathbf{in} \\ \qquad\qquad \mathbf{let}\ b = (\mathbf{let}\ y' = \mathrm{decide}\ ()\ \mathbf{in}\ \mathbf{if}\ y'\ \mathbf{then}\ \mathbf{return}\ 1\ \mathbf{else}\ \mathbf{return}\ 2)\ \mathbf{in}\ \mathbf{return}\ a - b)\end{array}$$

where $h \triangleq \{\mathbf{return}\ x_r \mapsto \mathbf{return}\ x_r, \mathrm{decide}(x, k) \mapsto \mathbf{let}\ r_t = k\ \mathbf{true}\ \mathbf{in}\ \mathbf{let}\ r_f = k\ \mathbf{false}\ \mathbf{in}\ \max\ r_t\ r_f\}$. As seen before, executing this program results in 19. Our system can assign the most precise type $\{z : \mathrm{int} \mid z = 19\}$ to this program. We now show the typing process to achieve this. In what follows, we write $\Gamma_{\widetilde{x}}$ for the typing context binding the variables $\widetilde{x}$ with some appropriate types $\widetilde{B}$. In particular, these variables have these base types: $x_r : \mathrm{int}$, $a : \mathrm{int}$, $b : \mathrm{int}$, $y : \mathrm{bool}$, and $y' : \mathrm{bool}$.

First, consider the types assigned to the clauses in the handler h. The return clause can be typed as $\Gamma_{x_r} \vdash \mathbf{return}\ x_r : \{z : \mathrm{int} \mid z = x_r\}$. The clause for decide can be typed as follows:

$$\Gamma \vdash \mathbf{let}\ r_t = k\ \mathbf{true}\ \mathbf{in}\ \mathbf{let}\ r_f = k\ \mathbf{false}\ \mathbf{in}\ \max\ r_t\ r_f : \{z : \mathrm{int} \mid \phi\} \tag{1}$$

where $\Gamma \triangleq X : (\mathrm{int}, \mathrm{bool}), x : \mathrm{unit}, k : (y : \mathrm{bool}) \to \{z : \mathrm{int} \mid X(z, y)\}$, $\phi \triangleq \forall r_t r_f. X(r_t, \mathbf{true}) \land X(r_f, \mathbf{false}) \implies z = \mathtt{max}(r_t, r_f)$, and $\mathtt{max}$ is a term-level function that returns the larger of given two integers. In this typing judgment, the predicate variable X abstracts over relationships between inputs y and outputs z of delimited continuations captured by calls to decide, and the refinement formula ϕ summarizes what the operation clause computes. Therefore, the operation signature Σ of the type of the handled computation, c_{body} in what follows, can be given as follows:

$$\Sigma \triangleq \{\mathrm{decide} : \forall X : (\mathrm{int}, \mathrm{bool}).(x : \mathrm{unit}) \to ((y : \mathrm{bool}) \to \{z : \mathrm{int} \mid X(z, y)\}) \to \{z : \mathrm{int} \mid \phi\}\}\ .$$

Therefore, we can conclude that the program is typable as desired by the following derivation

$$\dfrac{\begin{array}{c}\Gamma_{x_r} \vdash \mathbf{return}\ x_r : \{z : \mathrm{int} \mid z = x_r\} \qquad (\text{Judgment } (1)) \\ (\mathrm{I}) \ \ \vdash c_{body} : \Sigma \triangleright \mathrm{int} \;/\; (\forall x_r.\{z : \mathrm{int} \mid z = x_r\}) \Rightarrow \{z : \mathrm{int} \mid z = 19\}\end{array}}{\vdash \mathbf{with}\ h\ \mathbf{handle}\ c_{body} : \{z : \mathrm{int} \mid z = 19\}}\ (\text{T-Hndl})$$

if the premise (I) for c_{body} holds. We derive it by (T-LetIp), obtaining a derivation of the form

$$\dfrac{\begin{array}{c}(\mathrm{II}) \ \ \vdash (\mathbf{let}\ y = \mathrm{decide}\ ()\ \mathbf{in}\ \mathbf{if}\ y\ \cdots) : \Sigma \triangleright \mathrm{int} \;/\; (\forall a.C_1) \Rightarrow \{z : \mathrm{int} \mid z = 19\} \\ (\mathrm{III}) \ \ \Gamma_a \vdash \mathbf{let}\ b = \cdots\ \mathbf{in}\ \mathbf{return}\ a - b : \Sigma \triangleright \mathrm{int} \;/\; (\forall x_r.\{z : \mathrm{int} \mid z = x_r\}) \Rightarrow C_1\end{array}}{(\mathrm{I}) \ \ \vdash c_{body} : \Sigma \triangleright \mathrm{int} \;/\; (\forall x_r.\{z : \mathrm{int} \mid z = x_r\}) \Rightarrow \{z : \mathrm{int} \mid z = 19\}}\ (\text{T-LetIp})$$

for some type C_1.

We start by examining judgement (III) because its derivation gives the constraints to identify the type C_1. By (T-LetIp) again, we can derive

$$\dfrac{\begin{array}{c}(\mathrm{III}\text{-}1) \ \ \Gamma_a \vdash (\mathbf{let}\ y' = \mathrm{decide}\ ()\ \mathbf{in}\ \mathbf{if}\ y'\ \cdots) : \Sigma \triangleright \mathrm{int} \;/\; (\forall b.C_2) \Rightarrow C_1 \\ (\mathrm{III}\text{-}2) \ \ \Gamma_{a,b} \vdash \mathbf{return}\ a - b : \Sigma \triangleright \mathrm{int} \;/\; (\forall x_r.\{z : \mathrm{int} \mid z = x_r\}) \Rightarrow C_2\end{array}}{(\mathrm{III}) \ \ \Gamma_a \vdash \mathbf{let}\ b = \cdots\ \mathbf{in}\ \mathbf{return}\ a - b : \Sigma \triangleright \mathrm{int} \;/\; (\forall x_r.\{z : \mathrm{int} \mid z = x_r\}) \Rightarrow C_1}\ (\text{T-LetIp})$$

with the premises (III-1) and (III-2) and some type C_2. Judgment (III-2) is derivable by

$$\dfrac{\Gamma_{a,b} \vdash \textbf{return}\ a-b : \{z : \text{int} \mid z = a-b\} \qquad \text{(III-2-S)}\ \ \Gamma_{a,b} \vdash \{z : \text{int} \mid z = a-b\} <: \Sigma \triangleright \text{int}\ /\ (\forall x_r.\{z : \text{int} \mid z = x_r\}) \Rightarrow C_2}{\text{(III-2)}\ \ \Gamma_{a,b} \vdash \textbf{return}\ a-b : \Sigma \triangleright \text{int}\ /\ (\forall x_r.\{z : \text{int} \mid z = x_r\}) \Rightarrow C_2}\ \text{(T-Sub)}$$

with the derivation of the subtyping judgment (III-2-S):

$$\dfrac{\Gamma_{a,b} \vdash \Sigma <: \emptyset \qquad \Gamma_{a,b} \vdash \{z : \text{int} \mid z = a-b\} <: \text{int} \qquad \Gamma_{a,b} \mid \{z : \text{int} \mid z = a-b\} \vdash \square <: (\forall x_r.\{z : \text{int} \mid z = x_r\}) \Rightarrow C_2}{\text{(III-2-S)}\ \ \Gamma_{a,b} \vdash \{z : \text{int} \mid z = a-b\} <: \Sigma \triangleright \text{int}\ /\ (\forall x_r.\{z : \text{int} \mid z = x_r\}) \Rightarrow C_2}\ \text{(S-Comp)}$$

The first two subtyping premises are derivable trivially. We can derive the third one by letting $C_2 \triangleq \{z : \text{int} \mid z = a-b\}$ because:

$$\dfrac{\dfrac{\Gamma_{a,b}, x_r : \{z : \text{int} \mid z = a-b\}, z : \text{int} \vDash (z = x_r) \implies (z = a-b)}{\Gamma_{a,b}, x_r : \{z : \text{int} \mid z = a-b\} \vdash \{z : \text{int} \mid z = x_r\} <: \{z : \text{int} \mid z = a-b\}}\ \text{(S-Rfn)}}{\Gamma_{a,b} \mid \{z : \text{int} \mid z = a-b\} \vdash \square <: (\forall x_r.\{z : \text{int} \mid z = x_r\}) \Rightarrow \boxed{\{z : \text{int} \mid z = a-b\}}}\ \text{(S-Embed)}$$

where the grayed part is denoted by C_2 in the original premise. We note that our type inference algorithm automatically infers such a type by constraint solving (cf. Section 4). Next, judgment (III-1) is derivable by

$$\dfrac{\text{(III-1-1)}\ \ \Gamma_a \vdash \text{decide}\ () : \Sigma \triangleright \text{bool}\ /\ (\forall y'.C_3) \Rightarrow C_1 \qquad \text{(III-1-2)}\ \ \Gamma_{a,y'} \vdash \textbf{if}\ y'\ \cdots : \Sigma \triangleright \text{int}\ /\ (\forall b.C_2) \Rightarrow C_3}{\text{(III-1)}\ \ \Gamma_a \vdash (\textbf{let}\ y' = \text{decide}\ ()\ \textbf{in}\ \textbf{if}\ y'\ \cdots) : \Sigma \triangleright \text{int}\ /\ (\forall b.C_2) \Rightarrow C_1}\ \text{(T-LetIp)}$$

with the premises (III-1-1) and (III-1-2) and some type C_3. By letting $C_3 \triangleq \{z : \text{int} \mid z = (y'\ ?\ (a-1) : (a-2))\}$, we can derive judgment (III-1-2):

$$\dfrac{\Gamma_{a,y'} \vdash \textbf{if}\ y'\ \cdots : \{z : \text{int} \mid z = (y'\ ?\ 1 : 2)\} \qquad \Gamma_{a,y'} \vdash \{z : \text{int} \mid z = (y'\ ?\ 1 : 2)\} <: \Sigma \triangleright \text{int}\ /\ (\forall b.C_2) \Rightarrow C_3}{\text{(III-1-2)}\ \ \Gamma_{a,y'} \vdash \textbf{if}\ y'\ \cdots : \Sigma \triangleright \text{int}\ /\ (\forall b.C_2) \Rightarrow C_3}\ \text{(T-Sub)}$$

It is easy to see that the first typing premise holds. We can derive the second subtyping premise similarly to subtyping judgment (III-2-S), namely, by (S-Comp) with the following derivation for the subtyping on control effects:

$$\dfrac{\dfrac{\Gamma_{a,y'}, b : \{z : \text{int} \mid z = (y'\ ?\ 1 : 2)\} \vDash (z = a-b) \implies (z = (y'\ ?\ (a-1) : (a-2)))}{\Gamma_{a,y'}, b : \{z : \text{int} \mid z = (y'\ ?\ 1 : 2)\} \vdash C_2 <: C_3}\ \text{(S-Rfn)}}{\Gamma_{a,y'} \mid \{z : \text{int} \mid z = (y'\ ?\ 1 : 2)\} \vdash \square <: (\forall b.C_2) \Rightarrow C_3}\ \text{(S-Embed)}$$

Judgment (III-1-1) is derived by (T-Op), but for that, we need to instantiate the predicate variable X in the type scheme of decide in Σ with a predicate A such that the constraint $C_3 = \{z : \text{int} \mid A(z, y')\}$ imposed by (T-Op) is met. Let $A \triangleq \lambda(z, y).z = (y\ ?\ (a-1) : (a-2))$, which satisfies the constraint trivially. Then, by letting $C_1 \triangleq \{z : \text{int} \mid \phi\}[A/X]$, we have the following derivation:

$$\dfrac{\Gamma_a \vdash () : \text{unit}}{\text{(III-1-1)}\ \ \Gamma_a \vdash \text{decide}\ () : \Sigma \triangleright \text{bool}\ /\ (\forall y'.\{z : \text{int} \mid A(z, y')\}) \Rightarrow \{z : \text{int} \mid \phi\}[A/X]}\ \text{(T-Op)}$$

Finally, we examine judgment (II). It is derivable by

$$\dfrac{\text{(II-1)}\ \ \vdash \text{decide}\ () : \Sigma \triangleright \text{bool}\ /\ (\forall y.C_4) \Rightarrow \{z : \text{int} \mid z = 19\} \qquad \text{(II-2)}\ \ \Gamma_y \vdash \textbf{if}\ y\ \cdots : \Sigma \triangleright \text{int}\ /\ (\forall a.C_1) \Rightarrow C_4}{\text{(II)}\ \ \vdash (\textbf{let}\ y = \text{decide}\ ()\ \textbf{in}\ \textbf{if}\ y\ \cdots) : \Sigma \triangleright \text{int}\ /\ (\forall a.C_1) \Rightarrow \{z : \text{int} \mid z = 19\}}\ \text{(T-LetIp)}$$

with the premises (II-1) and (II-2) and some type C_4. Judgement (II-2) is derivable similarly to (III-1-2) by letting $C_4 \triangleq \{z : \text{int} \mid z = (y\ ?\ 9 : 19)\}$. For judgment (II-1), we instantiate the predicate variable X in the first call to decide with the predicate $A' \triangleq \lambda(z, y).z = (y\ ?\ 9 : 19)$. Then, we can derive the judgment by the following derivation:

$$\dfrac{\vdash \mathsf{decide}\ () : \Sigma \triangleright \mathsf{bool}\ /\ (\forall y.\{z : \mathsf{int} \mid A'(z,y)\}) \Rightarrow \{z : \mathsf{int} \mid \phi[A'/X]\} \qquad \vdash \Sigma \triangleright \mathsf{bool}\ /\ (\forall y.C_4) \Rightarrow \{z : \mathsf{int} \mid \phi[A'/X]\} <: \Sigma \triangleright \mathsf{bool}\ /\ (\forall y.C_4) \Rightarrow \{z : \mathsf{int} \mid z = 19\}}{(\text{II-1}) \quad \vdash \mathsf{decide}\ () : \Sigma \triangleright \mathsf{bool}\ /\ (\forall y.C_4) \Rightarrow \{z : \mathsf{int} \mid z = 19\}}\ (\text{T-Sub})$$

(note that $C_4 = \{z : \mathsf{int} \mid A'(z,y)\}$) where the first premise is derived by (T-Op) and the second one holds because the formula $\phi[A'/X]$ is semantically equivalent to the formula $z = 19$.

We note that the predicate variable in the type scheme of decide is important to typing this example. The delimited continuations captured by the two calls to decide behave differently. Namely, they respectively behave according to the predicates $A(u,v)$ and $A'(u,v)$ where u is the integer output given the Boolean input v. By using predicate variables, our type system gives a single type scheme to an operation that abstracts over such different behaviors.[4]

3.3.2 Example 2: State. We next revisit the second example from Section 2.1. Recall the example, which is the following program:

$$(\textbf{with}\ h\ \textbf{handle}\ \ (\mathsf{set}\ 3;\ \textbf{let}\ n = \mathsf{get}\ ()\ \textbf{in}\ \mathsf{set}\ 5;\ \textbf{let}\ m = \mathsf{get}\ ()\ \textbf{in}\ n + m))\ 0$$

where $h \triangleq \{x_r \mapsto \lambda s.x_r,\ \mathsf{set}(x,k) \mapsto \lambda s.k\ ()\ x,\ \mathsf{get}(x,k) \mapsto \lambda s.k\ s\ s\}$. For this example, we use the following syntactic sugars: $c_1;\ c_2 \triangleq \textbf{let}\ x = c_1\ \textbf{in}\ c_2$ (where x does not occur in c_2) and $\lambda x.v \triangleq \lambda x.\textbf{return}\ v$. Then, the program is in our language. This program uses two operations: set, which updates the state value, and get, which returns the current state value. The handling construct returns a function that maps any integer value to the value 8; arguments to the function are initial state values, but they are not used because the function begins by initializing the state. Applying the function to the initial state value 0, the whole program returns 8.

This program is expected to be of the type $\{z : \mathsf{int} \mid z = 8\}$. The rest of this section explains how the type system assigns this type to the program. First, the operation signature Σ for the handler h can be defined as follows:

$$\begin{aligned}\Sigma \triangleq \{&\mathsf{set} : \forall X : (\mathsf{int},\mathsf{int}).\ (x : \mathsf{int}) \to (\mathsf{unit} \to ((s : \mathsf{int}) \to \{z : \mathsf{int} \mid X(z,s)\})) \\ &\qquad\qquad \to ((s : \mathsf{int}) \to \{z : \mathsf{int} \mid X(z,x)\}), \\ &\mathsf{get} : \forall X : (\mathsf{int},\mathsf{int},\mathsf{int}).\ \mathsf{unit} \to ((y : \mathsf{int}) \to ((s : \mathsf{int}) \to \{z : \mathsf{int} \mid X(z,s,y)\})) \\ &\qquad\qquad \to ((s : \mathsf{int}) \to \{z : \mathsf{int} \mid X(z,s,s)\})\ \}\end{aligned}$$

Then, each sub-computation in the handled computation can be typed as follows:

$$\begin{aligned}&\vdash \mathsf{set}\ 3 : \Sigma \triangleright \mathsf{int}\ /\ (\forall\_.(s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = s + 5\}) \Rightarrow (s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = 3 + 5\} \\ &\vdash \mathsf{get}\ () : \Sigma \triangleright \mathsf{int}\ /\ (\forall n.(s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = n + 5\}) \Rightarrow (s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = s + 5\} \\ n : \mathsf{int} &\vdash \mathsf{set}\ 5 : \Sigma \triangleright \mathsf{int}\ /\ (\forall\_.(s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = n + s\}) \Rightarrow (s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = n + 5\} \\ n : \mathsf{int} &\vdash \mathsf{get}\ () : \Sigma \triangleright \mathsf{int}\ /\ (\forall m.(s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = n + m\}) \Rightarrow (s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = n + s\} \\ n : \mathsf{int}, m : \mathsf{int} &\vdash \textbf{return}\ n + m : \\ \Sigma \triangleright \{x_r &: \mathsf{int} \mid x_r = n + m\}\ /\ (\forall x_r.(s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = x_r\}) \Rightarrow (s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = n + m\}\end{aligned}$$

The first four judgments are derived by (T-Op) with appropriate instantiation of the type schemes of set and get. The last judgement is derived using (S-Embed) as in the first example. The type of the handled computation is derived from these computation types, taking the following form:

$$\Sigma \triangleright \mathsf{int}\ /\ (\forall x_r.(s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = x_r\}) \Rightarrow (s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = 8\}\ .$$

Therefore, by (T-Hndl), the type of the handling construct is $(s : \mathsf{int}) \to \{z : \mathsf{int} \mid z = 8\}$, and by (T-App), the type of the whole program is $\{z : \mathsf{int} \mid z = 8\}$ as promised.

[4]An alternative approach is to use intersection types (i.e., allow a set of types to be given to an operation). But, we find our approach more uniform and modular as it is able to give a single compact type scheme and enables operation signatures to be unaware of in which contexts operations are called.

3.3.3 Example 3: File Manipulation. Finally, we consider the last example in Section 2.1 that manipulates a specified file. Because the example uses nondeterministic while-loop constructs **while** $(\star)$ $\{c\}$, we informally extend our language with them.[5] The semantics of the while-loop constructs is given by the reduction rules **while** $(\star)$ $\{c\} \longrightarrow c$; **while** $(\star)$ $\{c\}$ and **while** $(\star)$ $\{c\} \longrightarrow$ **return** $()$, and the typing rule is given as follows:

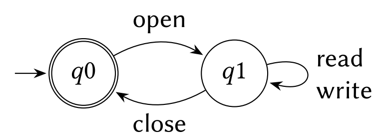
$$\frac{\Gamma \vdash c : \Sigma \triangleright \mathrm{unit} \ / \ C \Rightarrow C}{\Gamma \vdash \mathbf{while}\ (\star)\ \{c\} : \Sigma \triangleright \mathrm{unit} \ / \ C \Rightarrow C}\ (\text{T-Loop})$$

Note that it is easy to adapt the type safety to this extension.

Recall that the example for file manipulation is the following function:

$$v \triangleq \lambda x.\ \mathbf{while}\ (\star)\ \{\mathsf{open}\ x;\ \mathbf{while}\ (\star)\ \{\mathbf{let}\ y = \mathsf{read}()\ \mathbf{in}\ \mathsf{write}\ (y\text{\^{}}\texttt{"X"})\};\ \mathsf{close}\ ()\}\ .$$

The regular scheme stipulating the valid use of the file operations is (open (read | write)* close)*, which is equivalent to the automaton to the right.



Our idea to verify the correctness of the file manipulation is to encode the automaton states as program states, simulate the state transitions in the automaton by state-passing, and check that the file operations are used only in appropriate states. Let $Q0 \triangleq 0$ and $Q1 \triangleq 1$; they represent the automaton states $q0$ and $q1$, respectively. We suppose that an effect handler implements the file operations open , close, read, and write in a state-passing style for states $Q0$ and $Q1$. Then, the type scheme of each file operation can be given as an instance of the following template:

$$F(T_{\mathrm{in}}, T_{\mathrm{out}}, Q_{\mathrm{pre}}, Q_{\mathrm{post}}) \triangleq T_{\mathrm{in}} \to (T_{\mathrm{out}} \to (\{x : \mathrm{int} \mid x = Q_{\mathrm{post}}\} \to C)) \to (\{x : \mathrm{int} \mid x = Q_{\mathrm{pre}}\} \to C)$$

where the parameters T_{in} and T_{out} are the input and output types, respectively, of the operation, and Q_{pre} and Q_{post} are the states before and after, respectively, performing the operation. We do not specify the final answer type C concretely here because it is not important. Using this template, an operation signature Σ of the file operations is given as

$$\{\ \mathsf{open} : F(\mathrm{str}, \mathrm{unit}, Q0, Q1),\ \ \mathsf{close} : F(\mathrm{unit}, \mathrm{unit}, Q1, Q0),\ \ \mathsf{read} : F(\mathrm{unit}, \mathrm{str}, Q1, Q1),\ \ \mathsf{write} : F(\mathrm{str}, \mathrm{unit}, Q1, Q1)\ \}\ .$$

Note that the state transitions represented in Σ are matched with those in the automaton. Let

$$S(Q_{\mathrm{pre}}, Q_{\mathrm{post}}) \triangleq (\{x : \mathrm{int} \mid x = Q_{\mathrm{post}}\} \to C) \Rightarrow (\{x : \mathrm{int} \mid x = Q_{\mathrm{pre}}\} \to C)\ .$$

Given an effect handler h conforming to Σ and a computation c with control effect $S(n_{\mathrm{pre}}, n_{\mathrm{post}})$ for some n_{pre} and n_{post}, if a handling construct **with** h **handle** c is well typed, the body of h's return clause is typed at $\{x : \mathrm{int} \mid x = n_{\mathrm{post}}\} \to C$—i.e., the computation c terminates at the state n_{post}—and the handling construct **with** h **handle** c is typed at $\{x : \mathrm{int} \mid x = n_{\mathrm{pre}}\} \to C$—i.e., it requires n_{pre} as the initial state to start the computation c. Therefore, if $n_{\mathrm{pre}} = n_{\mathrm{post}} = Q0$, then it is guaranteed that the file operations are used in a valid manner. Furthermore, even if c is non-terminating, our type system can ensure that it does not use the file operations in an invalid manner. For example, suppose that c is a computation close $()$; Ω where Ω is a diverging computation. If it is well typed, its final answer type is $\{x : \mathrm{int} \mid x = Q1\} \to C$, which indicates that **with** h **handle** c requires $Q1$ as the initial state. It is clearly inconsistent with the above automation. As another instance, suppose that c involves a computation $\cdots$; close $()$; write "X"; $\cdots$. This is illegal because it tries to call write after close without open. Our type system rejects it because the initial answer type

[5] An alternative is to encode the while-loop constructs in our language by supposing that the termination of a while-loop construct is determined by some function parameter $f : \mathrm{unit} \to \mathrm{bool}$.

$\{x : \text{int} \mid x = Q0\} \rightarrow C$ of close () is not matched with the final answer type $\{x : \text{int} \mid x = Q1\} \rightarrow C$ of write "X" while they must be matched for the computation to be well typed.

We end this section by showing that the example function v can be typed at str $\rightarrow \Sigma \triangleright$ unit / $S(Q0, Q0)$, which means that v's body uses the file operations appropriately. Note that, for any typing context Γ' and file operation op, if op : $F(T_1, T_2, n_{\text{pre}}, n_{\text{post}}) \in \Sigma$ and $\Gamma' \vdash v : T_1$, then $\Gamma' \vdash \text{op}\ v : \Sigma \triangleright T_2 \ /\ S(n_{\text{pre}}, n_{\text{post}})$ by (T-Op). Let $\Gamma \triangleq x : \text{str}$. For the inner while-loop construct, we have the following typing derivation:

$$\dfrac{\dfrac{\Gamma \vdash \text{read}() : \Sigma \triangleright \text{str} \ /\ S(Q1, Q1) \qquad \Gamma, y : \text{str} \vdash \text{write}\ (y\verb|^"X"|) : \Sigma \triangleright \text{unit} \ /\ S(Q1, Q1)}{\Gamma \vdash \textbf{let}\ y = \text{read}()\ \textbf{in}\ \text{write}\ (y\verb|^"X"|) : \Sigma \triangleright \text{unit} \ /\ S(Q1, Q1)}\ (\text{T-LetIp})}{\Gamma \vdash \textbf{while}\ (\star)\ \{\textbf{let}\ y = \text{read}()\ \textbf{in}\ \text{write}\ (y\verb|^"X"|)\} : \Sigma \triangleright \text{unit} \ /\ S(Q1, Q1)}\ (\text{T-Loop})$$

Thus, the sub-computations of the outer while-loop construct can be typed as follows:

$$\begin{array}{ll} \Gamma \vdash \text{open}\ x & : \Sigma \triangleright \text{unit} \ /\ S(Q0, Q1) \\ \Gamma \vdash \textbf{while}\ (\star)\ \{\textbf{let}\ y = \text{read}()\ \textbf{in}\ \text{write}\ (y\verb|^"X"|)\} & : \Sigma \triangleright \text{unit} \ /\ S(Q1, Q1) \\ \Gamma \vdash \text{close}\ () & : \Sigma \triangleright \text{unit} \ /\ S(Q1, Q0)\ . \end{array}$$

where the control effects express how the state changes according to the operation calls. By (T-LetIp), (T-Loop), and (T-Fun), they then imply that v is typed at str $\rightarrow \Sigma \triangleright$ unit / $S(Q0, Q0)$ as desired.

3.4 Discussion

In this section, we discuss the current limitations and future extensions of our system.

3.4.1 Abstraction of Effects. Our type system has no mechanism for abstraction of effects. Therefore, if we cannot know possible effects of the handled computation in advance (e.g., as in $\lambda f.$**with** h **handle** $(f\ ())$, where the effects of the handled computation f () are determined by function parameter f), we have to fix its effects (both the operation signature and the control effect). A possible way to address this issue is to incorporate some mechanism to abstract effects. For operation signatures, effect polymorphism as in the existing effect systems for algebraic effects and handlers [Leijen 2017; Lindley et al. 2017], is a promising solution. However, adapting it to our system is not trivial. Effect polymorphism enables specifying a part of an operation signature as a parameter, and handling constructs implicitly forward operations in the parameter. The problem is that *our type system modifies the type schemes of forwarded operations* (see the supplementary material for detail). Therefore, even though the type schemes are involved in an operation signature parameter, we need to track how they are modified. We leave addressing this challenge for future work. For control effects, we conjecture that bounded polymorphism can be used to abstract control effects while respecting the necessary sub-effecting constraints.

3.4.2 Combination with Other Computational Effects. Algebraic effects and handlers are sometimes used with other computational effects. For example, when implementing a scheduler with algebraic effects and handlers, an imperative queue is often used to keep suspended continuations, like in an example from Multicore OCaml [2022]. Even though some computational effects can be simulated by algebraic effects and handlers themselves, it is often convenient to address them as primitive operations for efficiency. Our system does not support such primitive computational effects. It is left for future work to combine these features in one system.

3.4.3 Shallow Handlers. The handlers we adopt in this work are called *deep handlers* [Kammar et al. 2013], which is the most widely used variant. Another variant of algebraic effect handlers is *shallow handlers* [Hillerström and Lindley 2018], which we do not address in the present work. Just as deep handlers are related to shift0/reset0, shallow handlers are related to control0/prompt0 [Piróg et al. 2019]. Therefore, the type system for control0/prompt0 with ATM [Ishio and Asai 2022] may be adapted to develop a refinement type system for shallow handlers, as we have developed our refinement type system for deep handlers based on the type systems for shift0/reset0 with ATM [Materzok and Biernacki 2011; Sekiyama and Unno 2023].

3.4.4 Recursive Computation Types. Some useful programs with algebraic effect handlers are ill typed in our system due to the lack of support for recursive computation types. For example, consider the following program:

$$\mathbf{rec}(f, n).\mathbf{with}\ h\ \mathbf{handle}\ \mathbf{if}\ n = 0\ \mathbf{then}\ \mathsf{Err}\ \texttt{"error"}\ \mathbf{else}\ f\ (n-1)$$

where $h \triangleq \{\mathsf{Err}(msg, k) \mapsto \mathsf{Err}\ (\texttt{sprintf "called at \%d. \%s"}\ n\ msg)\}$. This recursive function handles the error in each function call, producing its own stack trace. It cannot be typed without recursive types because the type of the handled computations appears recursively as its answer type. To see this, assume that the type of the handled computation (i.e., the conditional branch) is assigned a type $\Sigma \triangleright T\ /\ C_1 \Rightarrow C_2$ (here we consider only simple types for simplicity). Then, the type of the handling construct (i.e., the body of the function) is C_2, which implies that the overall function has type $\mathsf{int} \to C_2$. And so, the recursive call to the function $f\ (n-1)$ also has type C_2. Then, the type C_2 should be a subtype of $\Sigma \triangleright T\ /\ C_1 \Rightarrow C_2$ since $f\ (n-1)$ is the else-branch of the conditional branch. However, we cannot derive $\Gamma \vdash C_2 <: \Sigma \triangleright T\ /\ C_1 \Rightarrow C_2$ (for some Γ) in our system because while the type on the left-hand side is C_2 itself, C_2 appears as the answer type in the control effect of the type on the right-hand side. On the other hand, using recursive types, we can give this function the following type (again, we consider only simple types for simplicity): $\mathsf{int} \to \mu\alpha.\Sigma_\alpha \triangleright T\ /\ T \Rightarrow \alpha$ where $\Sigma_\alpha \triangleq \{\mathsf{Err} : \mathsf{str} \to (T \to T) \to \alpha\}$ and T is an arbitrary value type. Type $\mu\alpha.C$ denotes a recursive computation type where the type variable α refers to the whole type itself. The control effect of this type is recursively nested, which reflects the fact that the handling construct is recursively nested due to the recursive call to the function.

3.4.5 Type Polymorphic Effect Operations. Consider the following program that evaluates to $[[21]]$:

$$\mathbf{with}\ \{x_r \mapsto x_r, \mathsf{wrap}((), k) \mapsto [k\ ()]\}\ \mathbf{handle}\ (\mathsf{wrap}\ (); \mathsf{wrap}\ (); 21)$$

This does not type-check in our current system because a type polymorphic operation signature like $\Sigma \triangleq \{\mathsf{wrap} : \forall\alpha.\mathsf{unit} \to (\mathsf{unit} \to \alpha\ /\ S) \to \alpha\ \mathsf{list}\ /\ S'\}$ is required. It is, however, easy to extend our type system to support type polymorphic operation signatures to handle such examples. Specifically, in the typing of operation clauses c_i in the (T-Hndl) rule, one would generalize type variables, and in the (T-Op) rule, one would instantiate type polymorphism.

3.4.6 Modularity (or Abstraction) versus Preciseness (or Concreteness). In our system, operation signatures are of the form $\mathsf{op}_i : T_i \to (T_i' \to C_i) \to C_i'$ where the types C_i and C_i' represent behavior of the effect handler. In other words, the signature reveals specific implementation details regarding effect handlers. This design, from our perspective of precise specification and verification, is valuable. Indeed, our type system can formally specify and verify the assume-guarantee-like contracts between the handler and operation-call sides. However, from the perspective of modularity and abstraction, this design choice is not the optimal one. In fact, one of the purposes of effect handlers is to abstract away the specifics so that one could later choose a different implementation.

To ensure that the handler implementation details do not leak in the operation signatures, one can introduce computation type polymorphism: The types C_i and C_i' in the operation signature above will be replaced by computation type variables, thus hiding the details. However, completely hiding the information of handler implementations in this way implies that we are not providing and verifying a detailed specification requirement for the handler implementations.

Practically speaking, rather than the two extremes, we believe that it is engineering-wise desirable to allow for a gradient between modularity (abstraction) and preciseness (concreteness) and to describe and verify types at the appropriate level of detail depending on the use case. Introducing all the polymorphisms discussed in this section might achieve this goal, but we plan to investigate whether it is indeed the case by specifying and verifying various real-world programs. In our view, the issue of how to describe types at an appropriate level of abstraction, as discussed above, is an important open problem not just for algebraic effects but for general control operators and, more broadly, for effectful computation.

4 IMPLEMENTATION

4.1 Description of Our Implementation

In this section, we describe our prototype implementation of a refinement type checking and inference system, RCaml[6]. It takes a program written in a subset of the OCaml 5 language (including algebraic data types, pattern matching, recursive functions, exceptions, mutable references[7], let-polymorphism, and effect handlers) and a refinement-type specification for the function of interest. It first (1) obtains an ML-typed AST of the program using OCaml's compiler library, (2) infers refinement-free operation signatures and control effects, (3) generates refinement constraints for the program and its specification as Constrained Horn Clauses (CHCs) (see e.g., the work of Bjørner et al. [2015]), and finally (4) solves these constraints to verify if the program satisfies the specification. The steps (3) and (4), where the refinement type checking is reduced to CHC solving, follow existing standard approaches such as those proposed by Rondon et al. [2008] and Unno and Kobayashi [2009]. The inference of (refinement-free) operation signatures is similar to that of record types using row variables, and is mutually recursive with the inference of control effects. It is based on the inference of control effects for shift0/reset0 [Materzok and Biernacki 2011]. As we split the steps of CHC generation and solving, we can use different solvers as the backend CHC solver depending on benchmarks. In this experiment, we used two kinds of CHC solvers: Spacer [Komuravelli et al. 2013] that is based on Property Directed Reachability (PDR) [Bradley 2011; Een et al. 2011], and PCSat [Unno et al. 2021] that is based on template-based CEGIS [Solar-Lezama et al. 2006; Unno et al. 2021] with Z3 [de Moura and Bjørner 2008] as an SMT solver.

Because inputs to the implementation are OCaml programs that are type-checked by OCaml's type checker which does not allow ATM, the underlying OCaml types corresponding to the answer types cannot be modified. However, as remarked before in Section 1, our aim is to verify *existing* programs with algebraic effects and handlers, and, as remarked before, our ARM, that allows only modification in the refinements, is useful for that purpose.

Our implementation supports several kinds of polymorphism. In addition to the standard let-polymorphism on types, it supports refinement predicate polymorphism. The implementation extends the formal system by allowing *bounded* predicate polymorphism in which abstracted predicates can be bounded by constraints on them, and further allows predicate-polymorphic types to be assigned to let-bound terms. However, because the implementation can infer predicate-polymorphic types only at let-bindings, we used a different approach, which we will discuss in Section 4.2, to simulate predicate polymorphism in operation signatures.

Another notable point is that our implementation deals with operations and exceptions uniformly. That is, exception raising is treated as an operation invocation and it can be handled by a certain kind of effect handlers which have clauses for exceptions (the exception clauses are included in the effect handlers of OCaml by default).

[6]available at https://github.com/hiroshi-unno/coar

[7]Strong updates [Foster et al. 2002] are not supported.

4.2 Evaluation

We performed a preliminary experiments to evaluate our method on some benchmark programs that use algebraic effect handlers. The benchmarks are based on example programs from Bauer and Pretnar [2015] and the repository of the Eff language [Pretnar 2022]. We gathered the effect handlers in those examples and created benchmark programs each of which uses one of the effect handlers. We also added a refinement type specification of the function of interest to each benchmark. (Other auxiliary functions are not given such extra information, and so their types are *inferred automatically* even for recursive functions.) Most benchmarks could be solved automatically without the annotations, but some need them as hints. We discuss the details at the end of this section. It is also notable that, although the examples presented in Section 3.3 focus on the specifications specialized in concrete, constant values such as $\{z : \text{int} \mid z = 19\}$ for Example 1, the benchmarks include programs that demonstrate that our type system and implementation can address more general specifications.For instance, the specification for the benchmark `choose-max-SAT.ml`, which is a general version of Example 1 where the constants 10, 20, 1, and 2 are replaced by parameters u, v, x, and y, respectively, of a function `main` to be verified, is as follows:

$$\vdash \texttt{main} : (u : \text{int}) \to (v : \{z{:}\text{int} \mid z \geq u\}) \to (x : \text{int}) \to (y : \{z{:}\text{int} \mid z \geq x\}) \to \{z{:}\text{int} \mid z = v - x\}$$

We refer to the supplementary material for the source code and the specifications of our benchmarks. All the experiments were conducted on Intel Xeon Platinum 8360Y, 256 GB RAM.

Table 1 shows the results of the evaluation. The files that are suffixed with `-SAT` are expected to result in "SAT", that is, the programs are expected to be typed with the refinement types given as their specifications. The other files (suffixed with `-UNSAT`) are expected to result in "UNSAT", that is, the programs are expected not to be typed with the given refinement types. For each program, we conducted verification in two configurations ((1) Spacer, and (2) PCSat). The field "time" indicates the time spent in the whole process of the verification. We set the timeout to 600 seconds. Our implementation successfully answered correct result for most programs. For instance, we show the benchmark `io-write-2-SAT.ml` as an example (where `@annot_MB` is an effect annotation written in the underlying OCaml type, explained in the last paragraph of this section):

```
let[@annot_MB "(unit -> ({Write: s} |> unit / s3 => s3)) -> unit * int list"]
  accumulate (body: unit -> unit) = match_with body () {
    retc = (fun v -> (v, []));  exnc = raise;
    effc = fun (type a) (e: a eff) -> match e with
      | Write x -> Some (fun (k: (a, _) continuation) ->
        let (v, xs) = continue k () in (v, x :: xs) ) }
let write_all l = accumulate (fun () ->
  let rec go li = match li with
    | [] -> () | s :: ss -> let _ = perform (Write s) in go ss
  in go l )
```

It iterates over a list `l` to pass its elements to the operation `Write`, and the handler for `Write` accumulates the passed elements into another list. It is checked against the following specification:

$$\vdash \texttt{write\_all} : \{z : \text{int list} \mid z \neq []\} \to \{z : \text{unit} \times \text{int list} \mid \forall u, v.\, z = (u, v) \Rightarrow v \neq []\}$$

That is, if the iterated list is not empty, the accumulated list is not, either. Our implementation successfully answered that `write_all` satisfies the specification, with the following inferred type:

$$(l : \{z : \text{int list} \mid z \neq []\}) \to \{z : \text{unit} \times \{z' : \text{int} \mid l \neq []\}\ \text{list} \mid \phi\}$$

where $\phi \triangleq \exists t : \text{int list}.(t = [] \lor z.2 \neq []) \land t \neq [] \land l \neq []$ and $z.2$ means the second element of the pair z. ARM is indispensable for this example because the initial answer type of the body of the

Table 1. Evaluation results

file name	Spacer		PCSat	
	result correct?	time (sec.)	result correct?	time (sec.)
amb-1-SAT.ml	Yes	0.55	Yes	15.30
amb-1-UNSAT.ml	Yes	0.72	Yes	63.62
amb-2-SAT.ml	Yes	2.31	Yes	31.48
amb-2-UNSAT.ml	Yes	2.26	-	timeout†
amb-3-SAT.ml	Yes	3.20	Yes	182.41
amb-3-simpl-SAT.ml	Yes	1.71	Yes	16.79
bfs-SAT.ml	No*1	1.67	-	timeout*1
bfs-UNSAT.ml	Yes	2.00	-	timeout†
bfs-simpl-SAT.ml	No*1	2.22	-	timeout*1
choose-all-SAT.ml	Yes	16.23	-	timeout†
choose-all-UNSAT.ml	Yes	12.56	-	timeout†
choose-max-SAT.ml	Yes	23.08	-	timeout†
choose-max-UNSAT.ml	Yes	15.97	-	timeout†
choose-sum-SAT.ml	Yes	1.54	-	timeout†
choose-sum-UNSAT.ml	Yes	7.99	Yes	15.00
deferred-1-SAT.ml	Yes	0.46	Yes	4.49
deferred-1-UNSAT.ml	Yes	0.27	Yes	4.09
deferred-2-SAT.ml	Yes	0.43	Yes	4.38
distribution-SAT.ml	Abort÷	-	-	timeout*2
distribution-UNSAT.ml	Abort÷	-	-	timeout*2
expectation-SAT.ml	Yes	0.51	Yes	7.25
expectation-UNSAT.ml	Yes	1.45	Yes	7.33
io-read-1-SAT.ml	Yes	0.43	Yes	13.90
io-read-1-UNSAT.ml	Yes	0.41	Yes	12.21
io-read-2-SAT.ml	Yes	0.56	Yes	21.10
io-read-3-SAT.ml	Yes	0.54	Yes	14.88
io-write-1-SAT.ml	Yes	0.32	Yes	8.48
io-write-1-UNSAT.ml	Yes	0.32	Yes	8.76
io-write-2-SAT.ml	Yes	0.46	Yes	11.33
io-write-2-UNSAT.ml	Yes	0.68	Yes	11.65
modulus-SAT.ml	Yes	14.23	Yes	11.89
modulus-UNSAT.ml	Yes	26.56	Yes	11.91
queue-1-SAT.ml	Yes	0.78	Yes	19.22
queue-1-UNSAT.ml	Yes	0.52	Yes	16.93
queue-2-SAT.ml	Yes	0.89	Yes	22.63
round-robin-SAT.ml	Yes	0.96	-	timeout†
round-robin-UNSAT.ml	Yes	0.73	-	timeout†
safe-div-1-SAT.ml	Abort÷	-	Yes	2.71
safe-div-1-UNSAT.ml	Abort÷	-	Yes	2.73
safe-div-2-SAT.ml	Abort÷	-	Yes	2.55
safe-div-2-UNSAT.ml	Abort÷	-	Yes	3.58
select-SAT.ml	-	timeout‡	Yes	13.28
select-UNSAT.ml	-	timeout‡	Yes	13.26
shift-SAT.ml	Yes	0.28	Yes	2.92
shift-UNSAT.ml	Yes	1.25	Yes	3.93
state-SAT.ml	-	timeout‡	Yes	33.69
state-UNSAT.ml	Yes	0.63	Yes	13.56
state-easy-SAT.ml	Yes	0.90	Yes	35.54
transaction-SAT.ml	-	timeout‡	Yes	15.36
transaction-UNSAT.ml	-	timeout‡	Yes	15.77
yield-SAT.ml	Yes	1.51	Yes	17.57
yield-UNSAT.ml	Yes	1.52	-	timeout†

function go should be $\{z : \text{unit} \times \text{int list} \mid z.2 = []\}$ (since it should be matched with the type of the return clause of the handler), while its final answer type should be $\{z : \text{unit} \times \text{int list} \mid z.2 \neq []\}$. We also present another interesting example (`queue-2-SAT.ml`) in detail in the supplementary material.

The benchmarks that were not verified correctly in both configurations are `bfs(-simpl)-SAT.ml` (marked with $*1$) and `distribution-(UN)SAT.ml` (marked with $*2$). They need some specific features which the implementation does not support. The formers need an invariant which states that there exists an element of a list that satisfies a certain property. The latter needs recursive predicates in the type of an integer list, which states a property about the sum of the elements of the list. These issues are orthogonal to the main contributions of this paper; they are about the expressiveness of the background theory used for refinement predicates, to which our novel refinement type system is agnostic. Also, `bfs(-simpl)-SAT.ml` uses mutable references which our implementation does not handle in a flow-sensitive manner (as mentioned in the footnote 7). One solution to this issue is to encode references with an effect handler as in Section 2.1, but our implementation does not do such encoding automatically. More advanced support for native effects including references is left for future work, as discussed in Section 3.4.

We discuss pros and cons between the two configurations. First, Spacer does not support division operator, and so it cannot verify some programs that use division (marked with ÷, aborting with the message "`Z3 Error: Uninterpreted 'div' in <null>`"). Also, some programs can be solved in one configuration but not in the other. Among those solved by Spacer but not by PCSat (marked with †), `round-robin-(UN)SAT.ml` timed out during the simplification of its constraints. For the remaining programs, their constraints tend to contain predicate variables that take a large number of arguments, which makes it hard for PCSat to find solutions. Conversely, the programs solved by PCSat but not by Spacer (marked with ‡) involve constraints where some predicate variables occur many times, which leads to complicated solutions that are difficult for Spacer to solve.

It is worth noting that our benchmarks do not rely on refinement type annotation in most places, even for recursive functions and recursive ADTs. However, a few kinds of annotations are still needed. First, as mentioned in Section 3.4, our type system does not support effect polymorphism. Therefore, we added effect annotations to function-type arguments which may perform operations when executed, as the one given to the benchmark `io-write-2-SAT.ml` using `@annot_MB`. These annotations are written in the underlying OCaml types, that is, we did not specify concrete refinements in the annotations. Second, we provided refinement type annotations for two small parts of `state-SAT.ml`, because otherwise it could not be verified within the timeout period in both configurations. Third, because our implementation infers predicate-polymorphic types only at let-bindings, we added *ghost parameters* to some operations and functions to infer precise refinement types of them which are not let-bound but need some abstraction of refinements. Ghost parameters are parameters which are used to express dependencies in dependent type checking, but have no impact on the dynamic execution of the program so they can be removed at runtime. In automated verification, completely inferring predicate variables requires higher-order predicate constraints, which are not expressible with CHC. Therefore, we provided ghost parameters to make it possible to reduce the verification to CHC solving. For example, the following is a part of `state-SAT.ml`:

```
let rec counter c =
    let i = perform (Lookup c) in
    if i = 0 then c else (perform (Update (c, i - 1)); counter (c + 1))
in counter 0
```

which is handled by a handler that simulates a mutable reference similar to that of Example 2 in Section 3.3.2. Here, we pass the variable c to the operation `Lookup` and `Update` as the ghost

Evaluation, Typing and subtyping rules $\boxed{c \longrightarrow c'}$ $\boxed{\Gamma \vdash c : \tau}$ $\boxed{\Gamma \vdash \tau_1 <: \tau_2}$

$$(c : \tau) \longrightarrow c \qquad (\Lambda\alpha.c)\ \tau \longrightarrow c[\tau/\alpha] \qquad (\Lambda\widetilde{X : \widetilde{B}}.c)\ \widetilde{A} \longrightarrow c[\widetilde{A/X}] \qquad \{(\mathrm{op}_i = v_i)_i\}\#\mathrm{op}_i \longrightarrow v_i$$

$$\frac{\Gamma \vdash c : \forall\alpha.\tau' \quad \Gamma \vdash \tau}{\Gamma \vdash c\ \tau : \tau'[\tau/\alpha]} \qquad \frac{(\Gamma \vdash \tau_{1i} <: \tau_{2i})_i}{\Gamma \vdash \{(\mathrm{op}_i : \tau_{1i})_i, (\mathrm{op}'_i : \tau'_i)_i\} <: \{(\mathrm{op}_i : \tau_{2i})_i\}}$$

$$\frac{\Gamma, \beta \vdash \tau_1[\tau/\alpha] <: \tau_2 \quad \Gamma, \beta \vdash \tau \quad \beta \notin fv(\forall\alpha.\tau_1)}{\Gamma \vdash \forall\alpha.\tau_1 <: \forall\beta.\tau_2}$$

Fig. 5. The operational semantics and the type system of the target language (excerpt).

parameter. In the formal system presented in Section 3.2 where predicate polymorphism is available in operation signatures, we can give `Update` the type

$$\forall X{:}(\mathrm{int}, \mathrm{int}).\ (x{:}\mathrm{int}) \to (\mathrm{unit} \to ((s{:}\mathrm{int}) \to \{z{:}\mathrm{int} \mid X(z, s)\})) \to ((s{:}\mathrm{int}) \to \{z{:}\mathrm{int} \mid X(z, x)\})$$

in the same way as Example 2 in Section 3.3.2, and instantiate the predicate variable X with $\lambda(z, s).z = c + 1 + s$ to correctly verify `state-SAT.ml`. On the other hand, in the implementation, since predicate polymorphism is not available in operation signatures, the handler needs to know the concrete predicate which replaces X. However, the predicate contains c, which the handler cannot know without receiving some additional information. Therefore, we need to add the ghost parameter c to `Update` (and the same for `Lookup`). This time we added them manually, but one possible approach for automating insertion of ghost parameters is to adopt the technique proposed by Unno et al. [2013]. We conjecture that a similar technique can be used for our purpose.

5 CPS TRANSFORMATION

5.1 Definitions and Properties

This section presents the crux of our CPS transformation that translate the language defined in Section 3 to a λ-calculus without effect handlers. Readers interested in the complete definitions of the target language and the CPS transformation are referred to the supplementary material.

The target language of the CPS transformation is a polymorphic λ-calculus with records and recursion. Its program and type syntax are defined as follows:

$$\begin{array}{rcl} v & ::= & x \mid p \mid \mathbf{rec}(f : \tau_1, x : \tau_2).c \mid \Lambda\widetilde{X : \widetilde{B}}.c \mid \{(\mathrm{op}_i = v_i)_i\} \mid \Lambda\alpha.c \\ c & ::= & v \mid c\ v \mid \mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 \mid c\ \widetilde{A} \mid v\#\mathrm{op} \mid c\ \tau \mid (c : \tau) \\ \tau & ::= & \{x : B \mid \phi\} \mid (x : \tau_1) \to \tau_2 \mid \forall\widetilde{X : \widetilde{B}}.\tau \mid \{(\mathrm{op}_i : \tau_i)_i\} \mid \alpha \mid \forall\alpha.\tau \end{array}$$

In the target language, values are not strictly separated from computations as those in the source language; for example, functions in function applications can be computations. The metavariables α and β range over type variables. Expressions $\Lambda\alpha.c$ and $c\ \tau$ are a type abstraction and application, respectively. Type polymorphism is introduced to express the pure control effect in the target language using *answer type polymorphism* [Thielecke 2003]. Expressions $\{(\mathrm{op}_i = v_i)_i\}$ and $v\#\mathrm{op}$ are a record literal and projection, respectively. We use operation names as record labels for the target language to encode handlers using records. Our CPS transformation produces programs with type annotations for proving bidirectional type-preservation. Recursive functions with type annotations and type ascriptions $(c : \tau)$ are used to annotate programs. We abbreviate $\mathbf{rec}(f : T_1, x : T_2).c$ to $\lambda x : T_2.c$ if f does not occur in c. Types are defined in a standard manner. Typing contexts Γ are extended to include type variables. The operational semantics is almost standard. Figure 5 shows

$$
\begin{array}{rcl}
\llbracket \{x : B \mid \phi\} \rrbracket & \triangleq & \{x : B \mid \phi\} \qquad \llbracket (x : T) \rightarrow C \rrbracket \triangleq (x : \llbracket T \rrbracket) \rightarrow \llbracket C \rrbracket \\
\llbracket \Sigma \triangleright T \ / \ (\forall x.C_1) \Rightarrow C_2 \rrbracket & \triangleq & \forall \_.\llbracket \Sigma \rrbracket \rightarrow ((x : \llbracket T \rrbracket) \rightarrow \llbracket C_1 \rrbracket) \rightarrow \llbracket C_2 \rrbracket \\
\llbracket \Sigma \triangleright T \ / \ \square \rrbracket & \triangleq & \forall \alpha.\llbracket \Sigma \rrbracket \rightarrow (\llbracket T \rrbracket \rightarrow \alpha) \rightarrow \alpha \\
\llbracket \{(\mathsf{op}_i : \forall X_i : \widetilde{B}_i.F_i)_i\} \rrbracket & \triangleq & \{(\mathsf{op}_i : \forall X_i : \widetilde{B}_i.\llbracket F_i \rrbracket^{\mathcal{F}})_i\} \\
\llbracket (x : T_1) \rightarrow ((y : T_2) \rightarrow C_1) \rightarrow C_2 \rrbracket^{\mathcal{F}} & \triangleq & (x : \llbracket T_1 \rrbracket) \rightarrow \llbracket ((y : T_2) \rightarrow C_1) \rrbracket \rightarrow \llbracket C_2 \rrbracket
\end{array}
$$

$$
\llbracket (\mathsf{op}^{\widetilde{A}}\ v)^{\Sigma \triangleright T / (\forall y.C_1) \Rightarrow C_2} \rrbracket \triangleq \overline{\Lambda}\alpha.\overline{\lambda}h : \llbracket \Sigma \rrbracket.\overline{\lambda}k : (y : \llbracket T \rrbracket \rightarrow \llbracket C_1 \rrbracket).h\#\mathsf{op}\ \widetilde{A}\ \llbracket v \rrbracket\ (\lambda y' : \llbracket T \rrbracket.k\ y')
$$

$$
\llbracket (\textbf{with } h \textbf{ handle } c)^{C} \rrbracket \triangleq \llbracket c \rrbracket \mathbin{\overline{@}} \llbracket C \rrbracket \mathbin{\overline{@}} \llbracket h^{ops} \rrbracket \mathbin{\overline{@}} \llbracket h^{ret} \rrbracket
$$

$$
\text{where} \begin{cases}
h = \{\textbf{return } x_r^{T_r} \mapsto c_r, (\mathsf{op}_i^{\widetilde{X_i : \widetilde{B}_i}}(x_i^{T_{x_i}}, k_i^{T_{k_i}}) \mapsto c_i)_i\} \\
\llbracket h^{ops} \rrbracket \triangleq \{(\mathsf{op}_i = \Lambda X_i : \widetilde{B}_i.\lambda x_i : \llbracket T_{x_i} \rrbracket.\lambda k_i : \llbracket T_{k_i} \rrbracket.\llbracket c_i \rrbracket)_i\} \\
\llbracket h^{ret} \rrbracket \triangleq \lambda x_r : \llbracket T_r \rrbracket.\llbracket c_r \rrbracket
\end{cases}
$$

Fig. 6. CPS transformation of types and expressions (excerpt).

four evaluation rules. Type ascriptions simply drop the ascribed type τ. Type applications substitute a given type τ for the bound type variable α. Predicate applications are similar. Record projections with op_i extract the associated field v_i. The type system is also standard, presented in Figure 5. We write $\Gamma \vdash \tau$ to state that all the free variables (including type and predicate ones) in the type τ are bound in the typing context Γ. The subtyping for record types allows supertypes to forget some fields in subtypes, and the types of each corresponding field in two record types to be in the subtyping relation (we deem record types, as well as records, to be equivalent up to permutation of fields). The subtyping rule for type polymorphism is a weaker variant of the containment rule for polymorphic types [Mitchell 1988]. It is introduced to emulate (S-Embed) in the target language.

We show the key part of the CPS transformation in Figure 6. The upper half presents the transformation of types. The transformation of value types is straightforward. Operation signatures are transformed into record types, which means that operation clauses in a handler are transformed into a record. The transformations of computation types indicate that computations are transformed into functions that receive two value parameters: handlers and continuations. If the control effect is pure, the answer types of computations become polymorphic in CPS. This treatment of control effects is different from that of Materzok and Biernacki [2011], who define CPS transformation for control effects in the simply typed setting. Their CPS transformation transforms, in our notation, a computation type $T \ / \ \square$ into the type $\llbracket T \rrbracket$, and a type $T \ / \ C_1 \Rightarrow C_2$ into the type $(\llbracket T \rrbracket \rightarrow \llbracket C_1 \rrbracket) \rightarrow \llbracket C_2 \rrbracket$ (note that they address neither operation signatures nor dependent typing). Because the latter takes continuations whereas the former does not, CPS transformation needs to know where pure computations are converted into impure ones (via subtyping). To address this issue, Materzok and Biernacki's CPS transformation focuses on typing derivations in the source language rather than expressions. However, because our aim is at reducing the typing of programs with algebraic effects and handlers to that of programs without them, we cannot assume typing derivations in the source language to be available. By treating two kinds of control effects uniformly using answer type polymorphism, our CPS transformation can focus only on expressions (with type annotations).

The lower half of Figure 6 shows the key cases of the transformation of expressions. We separate abstractions and applications in the target language into *static* and *dynamic* ones, as in the work of Hillerström et al. [2017], for proving the preservation of the operational semantics (Theorem 5.1). Redexes represented by static applications are known as *administrative redexes*, inserted and reduced at compile (CPS-transformed) time. By contrast, redexes represented by dynamic applications are reduced at run time because they originate in the source program. Constructors for static expressions are denoted by the overline notation, like $\overline{\lambda}$, $\overline{\Lambda}$, and $\overline{@}$. We use the "at" symbol explicitly as an infix operator of static applications for clarification. Non-overlined abstractions and applications are dynamic ones, which are treated as ordinary expressions. Also, for backward type-preservation (Theorem 5.3), we extend the source language with type annotations. For example, in an operation call $(\mathsf{op}^{\widetilde{A}}\ v)^{\Sigma \triangleright T / (\forall y.C_1) \Rightarrow C_2}$, $\widetilde{A}$ are predicates used to instantiate the type scheme of the operation op, and $\Sigma \triangleright T\ /\ (\forall y.C_1) \Rightarrow C_2$ is the type of the operation call op v. Without type annotations, CPS-transformed expressions may have a type that cannot be transformed back to a type in the source language. An operation call $(\mathsf{op}^{\widetilde{A}}\ v)^{\Sigma \triangleright T / (\forall y.C_1) \Rightarrow C_2}$ is transformed into a function that seeks the corresponding operation clause in a given handler and then applies it to a given sequence of predicates, argument, and continuation. Note that the continuation is in the η-expanded form because, for the preservation of the operational semantics, we need a dynamic lambda abstraction that corresponds to the continuation $\lambda y.$**with** h **handle** $K[$**return** $y]$ introduced in the rule (E-HndlOp) of the source language. An expression **with** h **handle** c is transformed into a function that applies the CPS-transformed handled computation to the record of the CPS-transformed operation clauses and the CPS-transformed return clause (because the return clause works as the continuation of c). The transformation preserves operational semantics bidirectionally in the following way:

Theorem 5.1 (simulation). *Let $\equiv_\beta$ be the smallest congruence relation over expressions in the target language that satisfies $(\overline{\lambda}x : \tau.c)\ \overline{@}\ v \equiv_\beta c[v/x]$ and $(\overline{\Lambda}\alpha.c)\ \overline{@}\ \tau \equiv_\beta c[\tau/\alpha]$. If $c \longrightarrow^*$ **return** v, then $[\![c]\!]\overline{@}\tau\overline{@}\{\}\overline{@}(\lambda x : \tau.x) \longrightarrow^+ v'$ for some v' such that $[\![v]\!] \equiv_\beta v'$. Also, if $[\![c]\!]\overline{@}\tau\overline{@}\{\}\overline{@}(\lambda x : \tau.x) \longrightarrow^+ v'$, then $c \longrightarrow^*$ **return** v and $[\![v]\!] \equiv_\beta v'$ for some v.*

(Note that τ can be any type since types are irrelevant to the operational semantics.) The first half states that if a computation c in the source language evaluates to a value-return of v, the transformed computation $[\![c]\!]$ applied to a type, an empty handler $\{\}$, and a trivial continuation $\lambda x : \tau.x$ evaluates to the transformed value $[\![v]\!]$. Similarly, the second half states the reverse direction.

Now, we state forward and backward type-preservation of the CPS transformation.

Theorem 5.2 (Forward type-preservation). *The following holds: (1) If $\Gamma \vdash v : T$ then $[\![\Gamma]\!] \vdash [\![v]\!] : [\![T]\!]$. (2) If $\Gamma \vdash c : C$ then $[\![\Gamma]\!] \vdash [\![c]\!] : [\![C]\!]$.*

Theorem 5.3 (Backward type-preservation). *The following holds: (1) If $\emptyset \vdash [\![v]\!] : \tau$, then there exists some T such that $\emptyset \vdash v : T$ and $\emptyset \vdash [\![T]\!] <: \tau$. (2) If $\emptyset \vdash [\![c]\!] : \tau$, then there exists some C such that $\emptyset \vdash c : C$ and $\emptyset \vdash [\![C]\!] <: \tau$.*

Theorem 5.3 is implied immediately by backward type preservation of the CPS transformation for *open* expressions. See the supplementary material for the statement for open expressions. Theorem 5.3 indicates that it is possible to reduce typechecking in our source language to that in a language without effect handlers. That is, if ones want to verify whether an expression c has type C, they can obtain the same result as the direct verification by first applying CPS transformation to c and C, and then checking whether $[\![c]\!]$ has type $[\![C]\!]$ with a refinement type verification tool that does not support algebraic effect handlers.

Type annotations in the source language are necessary to restrict the image of the transformation. Without them, a CPS-transformed program may be of a type τ that cannot be transformed to a type in the source language inversely (i.e., there exists no type C in the source language satisfying $[\![C]\!] = \tau$). For example, consider $\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.k\ 0$, the CPS form (without annotations) of expression **return** 0. Without annotations, we can pick arbitrary types as the type of h. Therefore, it can have type $\forall\alpha.\mathrm{bool} \to (\mathrm{int} \to \alpha) \to \alpha$. However, there is no type C in the source language such that

$[\![C]\!] = \forall\alpha.\mathrm{bool} \to (\mathrm{int} \to \alpha) \to \alpha$. Even worse, the source language has no type that is a *subtype* of the type of the CPS form since bool and record types are incomparable with each other. Another example is $\lambda x.\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.k\ x$, the CPS form (again, without annotations) of expression $\lambda x.\mathbf{return}\ x$. Its type can be $(\mathrm{int} \to \mathrm{int}) \to \forall\alpha.\{\} \to ((\mathrm{int} \to \mathrm{int}) \to \alpha) \to \alpha$, that is, x can be of type $\mathrm{int} \to \mathrm{int}$. However, there is no value type T in the source language such that $[\![T]\!]$ is a subtype of $\mathrm{int} \to \mathrm{int}$. Note that since a function type in the source language is in the form $(x : T_x) \to C$, the right hand side of the arrow in the CPS-transformed function type must be in the form $\forall\alpha.\{\cdots\} \to \cdots$, which does not match with int. Therefore, without type annotations, Theorem 5.3 does not hold.

While our formalization requires concrete refinement type annotations in the source language, actually we can relax this restriction by using predicate variables as placeholders instead of concrete refinements in type annotations. This is because type annotations are only for prohibiting occurrences of types with unintended *structures*, not for restricting refinements. Those predicate variables are instantiated after CPS transformation with concrete predicates inferred by generating and solving CHC constraints that contain these predicate variables from the CPS-transformed expression. Formally, by allowing occurrences of predicate variables in type annotations of both the source and target language, and introducing predicate variable substitution σ, we can state that $[\![\sigma(c)]\!] = \sigma([\![c]\!])$. This means that, for an expression c that is annotated with types containing predicate variables, both of the followings result in the same expression: (1) first instantiating the predicate variables in c with concrete refinements, and then CPS-transforming it (i.e., CPS-transforming the concretely-annotated expression), and (2) first CPS-transforming c, and then instantiating the predicate variables in the CPS-transformed expression with the concrete refinements. In other words, concrete refinements are irrelevant to the CPS transformation. This irrelevance is ensured by the fact that refinements can depend only on first-order values because it means that handler variables h and continuation variables k, which occur only in CPS-transformed expressions, cannot be used in instantiated refinements. The reason why we have defined the CPS transformation with concrete refinements is just to state Theorem 5.2 and Theorem 5.3.

5.2 Comparison between the Direct Verification and the Indirect Verification

In this section, we compare the direct verification using our refinement system presented in Section 3 with the indirect verification via the CPS transformation presented above. One of the differences is that the direct verification requires special support of verification tools for algebraic effect handlers, while the indirect one can be done by existing tools without such support. On the other hand, the indirect verification has some disadvantages. First, in most cases, CPS-transformed programs tend to be complicated and be in the forms quite different from the source programs. This complexity incurred in the indirect typechecking may lead to confusing error messages when the typechecking fails. Transforming the inferred complex types back to the types of the source language would be helpful, but it is unclear whether we can do this because the inferred types of the CPS-transformed expressions do not necessarily correspond to the CPS-transformed types of the source expressions, as stated in Section 5.1. By contrast, because the direct typechecking deals with the structures of the source programs as they are, error messages can be made more user-friendly. Second, our CPS transformation needs a non-negligible amount of type annotations—type annotations are necessary in let-expressions, conditional branches, and recursive functions as well as operation calls and handling constructs. In practice, it is desired to infer as many types as possible. However, it seems quite challenging to define a CPS transformation that enjoys backward type-preservation and needs no, or few, type annotations. One of the possible approaches for addressing type annotations in more automated way is to use the underlying simple type system of our refinement type system for algebraic effect handlers. As mentioned in Section 5.1, concrete refinements are not necessary for type annotations. Therefore, we can generate type annotations for an expression using its simple type inferred by the underlying type system.

We also compare these two approaches based on an experiment. We used some direct style (DS) programs (i.e., programs using algebraic effect handlers), and for each program, we applied our CPS transformation manually and ran the verification on both DS one and CPS one. Additionally, we also compared them with optimized CPS programs where administrative redexes were reduced. We used the same implementation as the one in Section 4 with the configuration of Spacer. We added annotations of source programs to only top-level, closed first-order expressions, but the correctness of the verification can be justified by the preservation of dynamic semantics.

Table 2 shows the results of the experiment. The columns "✓?" show whether the verification result is correct. The columns "time" are in seconds. Some programs have no big difference in verification time among the three variants, but there are two notable things. First, optimized CPS version of `choose-sum` took more time than the other versions. This seems because the size of the program became larger by the optimization. The CPS `choose-sum` program contains some branching expressions and each branch uses variables representing its continuation and the outer handler. By reducing administrative redexes in the program, these variables are instantiated with a concrete continuation and handler, that is, the continuation and handler are copied to each branch, which results in larger size of the program and its constraints generated during the verification. Second, CPS version of `io-read-2` could not be verified correctly. One possible reason is lack of support for higher-order predicate polymorphism. Since CPS programs explicitly pass around continuations, their types tend to be higher-order. Then, in some cases, higher-order predicate polymorphism becomes necessary by CPS transformation.

Table 2. Evaluation results of CPS transformation

program	DS		CPS		CPS (opt)	
	✓?	time	✓?	time	✓?	time
`amb-2`	Yes	1.30	Yes	1.32	Yes	0.91
`choose-easy`	Yes	0.26	Yes	0.27	Yes	0.22
`choose-sum`	Yes	2.18	Yes	1.79	Yes	12.87
`io-read-2`	Yes	0.66	No	1.29	No	0.62
`simple`	Yes	0.11	Yes	0.16	Yes	0.14

6 RELATED WORK

6.1 Algebraic Effects and Handlers

Algebraic effect handlers introduced by Plotkin and Pretnar [2013] turned out to be greatly expressive, which have inspired researchers and programming language designers and leads to a variety of implementations [Bauer and Pretnar 2015; Brady 2013; Kammar et al. 2013; Leijen 2017; Lindley et al. 2017; Sivaramakrishnan et al. 2021]. For advanced verification of algebraic effects and handlers, Ahman [2017] proposed a dependent type system for algebraic effects and handlers. Brady [2013] introduced algebraic effect handlers to Idris, a dependently typed programming language. In contrast to our system, these systems do not allow initial answer types to depend on values passed to continuations. Ahman and Plotkin [2015] investigated an algebraic treatment of computational effects with refinement types, but their language is not equipped with effect handlers. To our knowledge, there is no research focusing on refinement type systems with support for algebraic effect handlers and their implementations for automated verification.

Cong and Asai [2022] provided a type system with ATM for algebraic effect handlers in a simply typed setting. Compared with ours, their system is limited in a few points. First, it allows programs to use only one operation. Second, the operation can be invoked two or more times only when it is handled by an effect handler where the result types of the return and operation clause are the same. This limitation is particularly critical for our aim, program verification, because it means that there is no way to track the state of continuations that changes with the execution of programs. For instance, the examples presented in Section 3.3 cannot be verified under such a restriction because they include multiple calls to an operation and each call changes the state of continuations. Our type system has none of these limitations—it supports multiple operations and an unlimited number of calls to operations even under a handler with clauses of different types. The key idea of our system to allow such a handler is to introduce the abstraction of operation clauses over predicates. By this abstraction, our type system can represent how the same operation clause behaves differently under different continuations.

Our CPS transformation is based on Hillerström et al. [2017]. They defined a CPS transformation from a language with effect handlers but without dependent/refinement types, and proved that it enjoys forward type-preservation, but they, and others, such as Cong and Asai [2022], who studied CPS transformation for effect handlers, did not consider the backward direction. Their transformation also assumes that programs are fully annotated with types.

6.2 Type Systems for Other Delimited Control Operators

ATM was proposed by Danvy and Filinski [1990] to type more expressions with the delimited control operators shift/reset. Cong and Asai [2018] proposed a dependent type system for shift/reset, where initial answer types cannot depend on values passed to continuations. A type system with ATM for another set of delimited control operators shift0/reset0, is developed by Materzok and Biernacki [2011]. They proposed a new subtyping relation that allow lifting pure expressions to impure ones. Based on their work, Sekiyama and Unno [2023] proposed a refinement type system for shift0/reset0. Their type system utilizes ATM for reasoning about traces (sequences of events) precisely. In their system, initial answer types *can* depend on values passed to continuations. Our control effects are inspired by their work, but they use the dependency of control effects mainly for reasoning about traces while we use it for refining properties of values. Their target operators shift0/reset0 are closely related to our target operators, algebraic effect handlers [Forster et al. 2017; Piróg et al. 2019]. However, naively applying their approach to algebraic effect handlers does not enable precise verification. A critical difference between shift0/reset0 and algebraic effect handlers is that, while shift0/reset0 allows deciding the usage of captured delimited continuations per each call site of the continuation-capture operator shift0, algebraic effect handlers require all the calls to the same operation under a handler h to be interpreted by the same operation clause in h. This hinders precise verification of the use of continuations per each operation call. Our type system solves this problem by abstracting the type schemes of operations over predicates.

7 CONCLUSION

We developed a sound refinement type system for algebraic effects and handlers, which adopts the concept of ATM (especially, ARM) to capture how the use of effects and the handling of them influence the results of computations. This enables precise analysis of programs with algebraic effects and handlers. We also implemented the type checking and inference algorithm for a subset of OCaml 5 and demonstrated the usefulness of ARM. Additionally, we defined a bidirectionally-type-preserving CPS transformation from our language with effect handlers to the language without effect handlers. It enables the reuse of existing refinement type checkers to verify programs with effect handlers, but makes programs to be verified complicated and requires them to be fully annotated. One possible direction for future work is to incorporate temporal verification as in Sekiyama and Unno [2023] into algebraic effects and handlers. Also, it is interesting to apply ARM to other variants of effect handlers, such as lexically scoped effect handlers [Biernacki et al. 2020; Zhang and Myers 2019].

DATA-AVAILABILITY STATEMENT

Our artifact is available in the GitHub repository, at https://github.com/hiroshi-unno/coar. The experimental results shown in Table 1 and Table 2 can be reproduced by following the instructions in `popl24ae/README.md` of the repository.

ACKNOWLEDGMENTS

We are grateful to anonymous reviewers for their helpful and useful comments on the paper, especially regarding its presentation. We also thank Yiyang Guo and Kanaru Isoda for their contribution to our implementation. This work was supported by JSPS KAKENHI Grant Numbers JP19K20247, JP22K17875, JP20H00582, JP20H04162, JP22H03564, JP20H05703, JP20K20625, and JP22H03570 as well as JST CREST Grant Number JPMJCR21M3.

REFERENCES

Danel Ahman. 2017. Handling Fibred Algebraic Effects. *Proc. ACM Program. Lang.* 2, POPL, Article 7 (dec 2017), 29 pages. https://doi.org/10.1145/3158095

Danel Ahman and Gordon Plotkin. 2015. Refinement types for algebraic effects. In *Abstracts of the 21st Meeting 'Types for Proofs and Programs' (TYPES)*. Institute of Cybernetics, Tallinn University of Technology, 10–11.

Andrew W. Appel. 1992. *Compiling with Continuations*. Cambridge University Press. https://doi.org/10.1017/CBO9780511609619

Kenichi Asai. 2009. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation* 22, 3 (01 Sep 2009), 275–291. https://doi.org/10.1007/s10990-009-9049-5

Andrej Bauer. 2018. What is algebraic about algebraic effects and handlers? *CoRR* abs/1807.05923 (2018). arXiv:1807.05923 http://arxiv.org/abs/1807.05923

Andrej Bauer and Matija Pretnar. 2013. An Effect System for Algebraic Effects and Handlers. In *Algebra and Coalgebra in Computer Science*, Reiko Heckel and Stefan Milius (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–16.

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of Logical and Algebraic Methods in Programming* 84, 1 (2015), 108–123. https://doi.org/10.1016/j.jlamp.2014.02.001 Special Issue: The 23rd Nordic Workshop on Programming Theory (NWPT 2011) Special Issue: Domains X, International workshop on Domain Theory and applications, Swansea, 5-7 September, 2011.

Jesper Bengtson, Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Sergio Maffeis. 2011. Refinement Types for Secure Implementations. *ACM Trans. Program. Lang. Syst.* 33, 2, Article 8 (feb 2011), 45 pages. https://doi.org/10.1145/1890028.1890031

Dariusz Biernacki, Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2020. Binders by day, labels by night: effect instances via lexically scoped handlers. *Proc. ACM Program. Lang.* 4, POPL (2020), 48:1–48:29. https://doi.org/10.1145/3371116

Nikolaj Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, Vol. 9300. 24–51.

Aaron R. Bradley. 2011. SAT-Based Model Checking without Unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation* (Austin, TX, USA) *(VMCAI'11)*. Springer-Verlag, Berlin, Heidelberg, 70–87.

Edwin Brady. 2013. Programming and Reasoning with Algebraic Effects and Dependent Types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/2500365.2500581

Youyou Cong and Kenichi Asai. 2018. Handling Delimited Continuations with Dependent Types. *Proc. ACM Program. Lang.* 2, ICFP, Article 69 (jul 2018), 31 pages. https://doi.org/10.1145/3236764

Youyou Cong and Kenichi Asai. 2022. Understanding Algebraic Effect Handlers via Delimited Control Operators. In *Trends in Functional Programming - 23rd International Symposium, TFP 2022, Virtual Event, March 17-18, 2022, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 13401)*, Wouter Swierstra and Nicolas Wu (Eds.). Springer, 59–79. https://doi.org/10.1007/978-3-031-21314-4_4

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming* (Nice, France) *(LFP '90)*. Association for Computing Machinery, New York, NY, USA, 151–160. https://doi.org/10.1145/91556.91622

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS'08* (Budapest, Hungary, March29 – April 6), Vol. 4963. Springer Berlin Heidelberg, 337–340.

Niklas Een, Alan Mishchenko, and Robert Brayton. 2011. Efficient Implementation of Property Directed Reachability. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design* (Austin, Texas) *(FMCAD '11)*. FMCAD Inc, Austin, Texas, 125–134.

Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. 2017. On the Expressive Power of User-Defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proc. ACM Program. Lang.* 1, ICFP, Article 13 (aug 2017), 29 pages. https://doi.org/10.1145/3110257

Jeffrey S. Foster, Tachio Terauchi, and Alexander Aiken. 2002. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 1–12. https://doi.org/10.1145/512529.512531

Timothy S. Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, David S. Wise (Ed.). ACM, 268–277. https://doi.org/10.1145/113445.113468

Daniel Hillerström and Sam Lindley. 2018. Shallow Effect Handlers. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 415–435.

Daniel Hillerström, Sam Lindley, Robert Atkey, and K. C. Sivaramakrishnan. 2017. Continuation Passing Style for Effect Handlers. In *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 84)*, Dale Miller (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 18:1–18:19. https://doi.org/10.4230/LIPIcs.FSCD.2017.18

Chiaki Ishio and Kenichi Asai. 2022. Type System for Four Delimited Control Operators. In *Proceedings of the 21st ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences* (Auckland, New Zealand) *(GPCE 2022)*. Association for Computing Machinery, New York, NY, USA, 45–58. https://doi.org/10.1145/3564719.3568691

Ohad Kammar, Sam Lindley, and Nicolas Oury. 2013. Handlers in Action. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming* (Boston, Massachusetts, USA) *(ICFP '13)*. Association for Computing Machinery, New York, NY, USA, 145–158. https://doi.org/10.1145/2500365.2500590

Ohad Kammar and Matija Pretnar. 2017. No value restriction is needed for algebraic effects and handlers. *Journal of Functional Programming* 27 (2017), e7. https://doi.org/10.1017/S0956796816000320

Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke. 2013. Automatic Abstraction in SMT-Based Unbounded Software Model Checking. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) *(CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 846–862.

Daan Leijen. 2014. Koka: Programming with Row Polymorphic Effect Types. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014 (EPTCS, Vol. 153)*, Paul Blain Levy and Neel Krishnaswami (Eds.). 100–126. https://doi.org/10.4204/EPTCS.153.8

Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 486–499. https://doi.org/10.1145/3009837.3009872

PaulBlain Levy, John Power, and Hayo Thielecke. 2003. Modelling environments in call-by-value programming languages. *Information and Computation* 185, 2 (2003), 182–210. https://doi.org/10.1016/S0890-5401(03)00088-9

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages* (Paris, France) *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 500–514. https://doi.org/10.1145/3009837.3009897

Marek Materzok and Dariusz Biernacki. 2011. Subtyping Delimited Continuations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming* (Tokyo, Japan) *(ICFP '11)*. Association for Computing Machinery, New York, NY, USA, 81–93. https://doi.org/10.1145/2034773.2034786

John C. Mitchell. 1988. Polymorphic type inference and containment. *Information and Computation* 76, 2 (1988), 211–249. https://doi.org/10.1016/0890-5401(88)90009-0

Multicore OCaml. 2022. *OCaml effects examples.* Retrieved November 5, 2022 from https://github.com/ocaml-multicore/effects-examples

Yoji Nanjo, Hiroshi Unno, Eric Koskinen, and Tachio Terauchi. 2018. A Fixpoint Logic and Dependent Effects for Temporal Property Verification. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, Anuj Dawar and Erich Grädel (Eds.). ACM, 759–768. https://doi.org/10.1145/3209108.3209204

Maciej Piróg, Piotr Polesiuk, and Filip Sieczkowski. 2019. Typed Equivalence of Effect Handlers and Delimited Control. In *4th International Conference on Formal Structures for Computation and Deduction (FSCD 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 131)*, Herman Geuvers (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 30:1–30:16. https://doi.org/10.4230/LIPIcs.FSCD.2019.30

Gordon Plotkin. 1975. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science* 1, 2 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

Gordon Plotkin and John Power. 2003. Algebraic Operations and Generic Effects. *Applied Categorical Structures* 11, 1 (01 Feb 2003), 69–94. https://doi.org/10.1023/A:1023064908962

Gordon Plotkin and Matija Pretnar. 2009. Handlers of Algebraic Effects. In *Programming Languages and Systems*, Giuseppe Castagna (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 80–94.

Gordon D. Plotkin and John Power. 2001. Adequacy for Algebraic Effects. In *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2030)*, Furio Honsell and Marino Miculan (Eds.). Springer, 1–24. https://doi.org/10.1007/3-540-45315-6_1

Gordon D Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science* Volume 9, Issue 4 (Dec. 2013). https://doi.org/10.2168/LMCS-9(4:23)2013

Matija Pretnar. 2015. An Introduction to Algebraic Effects and Handlers. Invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35. https://doi.org/10.1016/j.entcs.2015.12.003 The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

Matija Pretnar. 2022. *Eff*. Retrieved November 5, 2022 from https://github.com/matijapretnar/eff

Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. 2008. Liquid Types. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 159–169. https://doi.org/10.1145/1375581.1375602

Taro Sekiyama and Hiroshi Unno. 2023. Temporal Verification with Answer-Effect Modification: Dependent Temporal Type-and-Effect System with Delimited Continuations. *Proc. ACM Program. Lang.* 7, POPL, Article 71 (jan 2023), 32 pages. https://doi.org/10.1145/3571264

KC Sivaramakrishnan, Stephen Dolan, Leo White, Tom Kelly, Sadiq Jaffer, and Anil Madhavapeddy. 2021. Retrofitting Effect Handlers onto OCaml. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 206–221. https://doi.org/10.1145/3453483.3454039

Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 404–415. https://doi.org/10.1145/1168857.1168907

Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. 2016. Dependent types and multi-monadic effects in F. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM, 256–270. https://doi.org/10.1145/2837614.2837655

Tachio Terauchi. 2010. Dependent types from counterexamples. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, Manuel V. Hermenegildo and Jens Palsberg (Eds.). ACM, 119–130. https://doi.org/10.1145/1706299.1706315

Hayo Thielecke. 2003. From control effects to typed continuation passing. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, Alex Aiken and Greg Morrisett (Eds.). ACM, 139–149. https://doi.org/10.1145/604131.604144

Hiroshi Unno and Naoki Kobayashi. 2009. Dependent Type Inference with Interpolants. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming* (Coimbra, Portugal) *(PPDP '09)*. Association for Computing Machinery, New York, NY, USA, 277–288. https://doi.org/10.1145/1599410.1599445

Hiroshi Unno, Yuki Satake, and Tachio Terauchi. 2018. Relatively complete refinement type system for verification of higher-order non-deterministic programs. *Proc. ACM Program. Lang.* 2, POPL (2018), 12:1–12:29. https://doi.org/10.1145/3158100

Hiroshi Unno, Tachio Terauchi, and Naoki Kobayashi. 2013. Automating Relatively Complete Verification of Higher-Order Functional Programs. *SIGPLAN Not.* 48, 1 (jan 2013), 75–86. https://doi.org/10.1145/2480359.2429081

Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 2021. Constraint-Based Relational Verification. In *CAV'21*. Springer Berlin Heidelberg, 742–766.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 269–282. https://doi.org/10.1145/2628136.2628161

Panagiotis Vekris, Benjamin Cosman, and Ranjit Jhala. 2016. Refinement types for TypeScript. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 310–325. https://doi.org/10.1145/2908080.2908110

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 214–227. https://doi.org/10.1145/292540.292560

Yizhou Zhang and Andrew C. Myers. 2019. Abstraction-safe effect handlers via tunneling. *Proc. ACM Program. Lang.* 3, POPL (2019), 5:1–5:29. https://doi.org/10.1145/3290318

He Zhu and Suresh Jagannathan. 2013. Compositional and Lightweight Dependent Type Inference for ML. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7737)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 295–314. https://doi.org/10.1007/978-3-642-35873-9_19

A Typing Rule for Operation Forwarding

The typing rule for handling constructs presented in Section 3.2 of the main paper assumes that a handler covers all the operations performed by the handled expression. In this section, we present another typing rule for handling constructs to allow *operation forwarding*, that is, allow unhandled operations to be forwarded to outer handlers automatically. The idea of the typing rule is simple: we derive it from an implementation of operation forwarding. As mentioned in Section 3.1 of the main paper, operation forwarding can be implemented in a calculus without forwarding by adding to a handler an operation clause $\mathsf{op}(x,k) \mapsto \mathbf{let}\ y = \mathsf{op}\ x\ \mathbf{in}\ k\ y$ for each forwarded operation op. Therefore, we can derive the new typing rule from the typing of the added clauses. The following is the thus derived new typing rule for handling constructs which natively supports operation forwarding:

$$\frac{\begin{array}{c} h = \{\mathbf{return}\ x_r \mapsto c_r, (\mathsf{op}_i(x_i,k_i) \mapsto c_i)_i\} \quad \Gamma \vdash c : \Sigma \triangleright T \ /\ (\forall x_r.C_1) \Rightarrow C_2 \\ \Gamma, x_r : T \vdash c_r : C_1 \quad \left(\Gamma, \widetilde{X_i : \widetilde{B}_i}, x_i : T_{1i}, k_i : (y_i : T_{2i}) \to C_{1i} \vdash c_i : C_{2i}\right)_i \\ \left(\Sigma \ni \mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : T_{1i}) \to ((y_i : T_{2i}) \to C_{1i}) \to C_{2i}\right)_i \qquad \mathit{Ops}_{\mathrm{fwd}} = \mathit{dom}(\Sigma) \setminus \mathit{dom}(h) \\ \left(\begin{array}{l} \Sigma \ni \mathsf{op} : \forall \widetilde{X^{\mathsf{op}} : \widetilde{B^{\mathsf{op}}}}.(x^{\mathsf{op}} : T_1^{\mathsf{op}}) \to \\ \quad ((y^{\mathsf{op}} : T_2^{\mathsf{op}}) \to \Sigma' \triangleright T_0^{\mathsf{op}} \ /\ (\forall z^{\mathsf{op}}.C_0^{\mathsf{op}}) \Rightarrow C_1^{\mathsf{op}}) \to \Sigma' \triangleright T_0^{\mathsf{op}} \ /\ (\forall z^{\mathsf{op}}.C_0^{\mathsf{op}}) \Rightarrow C_2^{\mathsf{op}} \\ \Sigma' \ni \mathsf{op} : \forall \widetilde{X^{\mathsf{op}} : \widetilde{B^{\mathsf{op}}}}.(x^{\mathsf{op}} : T_1^{\mathsf{op}}) \to ((y^{\mathsf{op}} : T_2^{\mathsf{op}}) \to C_1^{\mathsf{op}}) \to C_2^{\mathsf{op}} \quad y^{\mathsf{op}} \notin C_0^{\mathsf{op}} \setminus \{z^{\mathsf{op}}\} \end{array}\right)_{\mathsf{op} \in \mathit{Ops}_{\mathrm{fwd}}} \end{array}}{\Gamma \vdash \mathbf{with}\ h\ \mathbf{handle}\ c : C_2}$$

where $\mathit{dom}(\Sigma)$ denotes the set of the operations associated by Σ and $\mathit{dom}(h)$ denotes the set of the operations handled by h, that is, the set $\{(\mathsf{op}_i)_i\}$. The first two lines are the same as (T-Hndl). The third line is also similar to the last premise of (T-Hndl), but here Σ is allowed to contain operations other than those handled by h. $\mathit{Ops}_{\mathrm{fwd}}$ is exactly the set of the unhandled (i.e., forwarded) operations. The last part is the requirement for the forwarded operations, which can be obtained from the typing derivations of $\mathsf{op}(x,k) \mapsto \mathbf{let}\ y = \mathsf{op}\ x\ \mathbf{in}\ k\ y$ as follows. When we simulate the operation forwarding with the explicit clause, the operation call $\mathsf{op}\ x$ in the clause is handled by an immediate outer handler (we denote it by h' in what follows). Therefore, its operation signature is different from Σ; in fact, it corresponds to Σ' in the rule. Also, the answer types of the original operation calls of op (i.e., the answer types of the operation calls of op in the handled computation c) should have Σ' as their operation signatures, because the final answer type corresponds to the type of the handling construct, which is handled by the immediate outer handler h'. Therefore, the types of the forwarded operations in Σ contains Σ' in their answer types. In addition, the types T_0^{op}, T_1^{op}, T_2^{op}, C_0^{op}, C_1^{op}, and C_2^{op} appear multiple times in Σ and Σ', restricting the type schemes of the operations in $\mathit{Ops}_{\mathrm{fwd}}$. This restriction can be understood as follows. First, assume that the original operation call of op in c has the operation signature Σ such that

$$\Sigma \ni \mathsf{op} : T_1^{\mathsf{op}} \to (T_2^{\mathsf{op}} \to \Sigma' \triangleright T_0^{\mathsf{op}} \ /\ C_0^{\mathsf{op}} \Rightarrow C_1^{\mathsf{op}}) \to \Sigma' \triangleright T_{0A}^{\mathsf{op}} \ /\ C_{0A}^{\mathsf{op}} \Rightarrow C_2^{\mathsf{op}}$$

for some T_1^{op}, T_2^{op}, T_0^{op}, C_0^{op}, C_1^{op}, T_{0A}^{op}, C_{0A}^{op}, C_2^{op}, and Σ', under a context Γ. Here we consider only simple types for simplicity, but a similar argument can be made for dependent and refinement types by appropriately naming the variables like in the rule above. Note that its answer types have Σ' as described earlier, and that we do not impose the restriction yet. From the assumption, the clause $\mathbf{let}\ y = \mathsf{op}\ x\ \mathbf{in}\ k\ y$ should be typed under the context $\Gamma, x : T_1^{\mathsf{op}}, k : T_2^{\mathsf{op}} \to \Sigma' \triangleright T_0^{\mathsf{op}} \ /\ C_0^{\mathsf{op}} \Rightarrow C_1^{\mathsf{op}}$. Then, the input type of op in the clause should be the type of x, namely, T_1^{op}, and the output type of op should be the type of the variable y, which turns out to be T_2^{op} from the type of k. Therefore, the operation signature Σ' for $\mathsf{op}\ x$ should contain $\mathsf{op} : T_1^{\mathsf{op}} \to (T_2^{\mathsf{op}} \to C_{1A}^{\mathsf{op}}) \to C_{2A}^{\mathsf{op}}$ for some C_{1A}^{op} and C_{2A}^{op} . Then, according to the typing rules for operation calls and let-expressions, it is required that $C_{1A}^{\mathsf{op}} = C_1^{\mathsf{op}}$, and the type of $\mathbf{let}\ y = \mathsf{op}\ x\ \mathbf{in}\ k\ y$ is $\Sigma' \triangleright T_0^{\mathsf{op}} \ /\ C_0^{\mathsf{op}} \Rightarrow C_{2A}^{\mathsf{op}}$. Finally, since the type of the clause corresponds to the final answer type of the operation op in Σ (which is $\Sigma' \triangleright T_{0A}^{\mathsf{op}} \ /\ C_{0A}^{\mathsf{op}} \Rightarrow C_2^{\mathsf{op}}$ from the assumption), it should satisfy $T_0^{\mathsf{op}} = T_{0A}^{\mathsf{op}}$, $C_0^{\mathsf{op}} = C_{0A}^{\mathsf{op}}$, and $C_{2A}^{\mathsf{op}} = C_2^{\mathsf{op}}$.

B Detailed explanation of the benchmark

In this section, we present the result of the verification of the benchmark `queue-2-SAT.ml` as an example. The following is the main part of the program of `queue-2-SAT.ml`:

```
let[@annot_MB "int list ->
    (unit -> ({Get_next: s1, Add_to_queue: s2} |> int option / s => s)) ->
    int option"]
```

```
  queue initial (body :unit -> int option) =
    match_with body () {
      retc = (fun x -> (fun _ -> x));
      exnc = raise;
      effc = fun (type a) (e: a eff) -> match e with
        | Get_next _ctx -> Some (fun (k: (a, _) continuation) ->
            (fun queue -> match queue with
              | [] -> continue k None []
              | hd::tl -> continue k (Some hd) tl) )
        | Add_to_queue v -> Some (fun (k: (a, _) continuation) ->
            (fun queue -> continue k () (queue @ [v])) )
    } initial

let main init =
  queue init (fun () ->
    perform (Add_to_queue 42);
    let _ = perform (Get_next 1(*dummy*)) in
    perform (Get_next 2(*dummy*)) )
```

This program uses two operations `Get_next` and `Add_to_queue`, which are used to dequeue and enqueue elements respectively. The function `queue` manages the queue. It receives an initial queue `initial` and the function `body`, handling the operations performed in `body` in the state-passing manner to simulate the behavior of the queue. The first three lines of the program are the underlying simple type annotation, which tells the function `queue` that the argument `body` may perform the operations `Get_next` and `Add_to_queue` and that its control effect is impure. This annotaion is necessary because our implementation does not support effect polymorphism as mentioned in Section 4 of the main paper. The main function `main` first enqueue one element, and then try to dequeue twice (`Get_next` returns `None` when the queue is empty). Note that we added a ghost parameter `_ctx` to `Get_next`, which is used to distinguish its two occurrences. We give `1` to the first occurrence of `Get_next`, and `2` to the second. This ghost parameter is crucial for the precise verification of this program, described later in this section.

We defined the following refinement type as the specification for the main function `main` (here after, we abbreviate the type int list and int option as ilist and iopt respectively):

$$\{z : \text{ilist} \mid z \neq []\} \to \{z : \text{iopt} \mid z \neq \texttt{None}\}$$

That is, if the queue is initially not empty, the last dequeue should return some value.

By running the verification of the program with the specification, our implementation returns "SAT" as shown in Table 1 in the main paper, that is, the function `main` certainly has the type given as the specification. Let us investigate more detail by seeing the inferred type of the function `queue`:

$$\begin{aligned}
&(\mathit{init} : \{z : \text{ilist} \mid z \neq []\}) \\
&\to (\text{unit} \to \Sigma \triangleright \text{iopt} \,/\, (\forall x.(\text{ilist} \to \{z : \text{iopt} \mid \phi_1\})) \Rightarrow (\{z : \text{ilist} \mid \phi_2\} \to \{z : \text{iopt} \mid z \neq \texttt{None}\})) \\
&\to \{z : \text{iopt} \mid z \neq \texttt{None}\}
\end{aligned}$$

$$\begin{aligned}
\text{where } \Sigma \stackrel{\text{def}}{=} \{&\texttt{Add_to_queue} : \text{int} \to (\text{unit} \to \\
&\quad ((q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{41}\})) \to (\{z : \text{ilist} \mid \phi_2\} \to \{z : \text{iopt} \mid z \neq \texttt{None}\}), \\
&\texttt{Get_next} : (\mathit{ctx} : \text{int}) \to ((y : \text{iopt}) \to \\
&\quad ((q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{31} \wedge \phi_{32}\})) \to ((q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{41} \wedge \phi_{42}\})\}
\end{aligned}$$

$$\begin{aligned}
\phi_1 &\stackrel{\text{def}}{=} \text{isSome}(x) \Rightarrow z \neq \texttt{None} & \phi_2 &\stackrel{\text{def}}{=} \mathit{init} \neq [] \Rightarrow z \neq [] \\
\phi_{31} &\stackrel{\text{def}}{=} \text{isCons}(q) \wedge \text{isSome}(y) \Rightarrow z \neq \texttt{None} & \phi_{32} &\stackrel{\text{def}}{=} \text{isSome}(y) \wedge \mathit{ctx} \geq 2 \Rightarrow z \neq \texttt{None} \\
\phi_{41} &\stackrel{\text{def}}{=} \text{isCons}(q) \wedge \text{isCons}(\text{tail}(q)) \Rightarrow z \neq \texttt{None} & \phi_{42} &\stackrel{\text{def}}{=} \text{isCons}(q) \wedge \mathit{ctx} \geq 2 \Rightarrow z \neq \texttt{None}
\end{aligned}$$

where isSome(x) holds if $x = \texttt{Some}\ v$ for some v, isCons(x) holds if $x = v{::}w$ for some v and w, and tail(x) returns the tail of the list x. In the operation signuature, we can find that `Add_to_queue` changes the answer type from $(q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{41}\}$ to $\{z : \text{ilist} \mid \phi_2\} \to \{z : \text{iopt} \mid z \neq \texttt{None}\}$. Therefore, `perform (Add_to_queue 42)` can be given the control effect

$$(\forall \_.((q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{41}\})) \Rightarrow (\{z : \text{ilist} \mid \phi_2\} \to \{z : \text{iopt} \mid z \neq \texttt{None}\})\ .$$

Similarly, in the operation signuature, `Get_next` changes the answer type from $(q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{31} \wedge \phi_{32}\}$ to $(q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{41} \wedge \phi_{42}\}$. Here, since the refinements of these answer types contain a condition on ctx, their truth depend on whether $ctx = 1$ (< 2) or $ctx = 2$ (≥ 2). This enables assigning different control effects (i.e., different ARM) to each occurrence of `Get_next` depending on the context. Namely, `perform (Get_next 1)` can be given the control effect

$$(\forall y.(q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{31}\}) \Rightarrow (q : \text{ilist}) \to \{z : \text{iopt} \mid \phi_{41}\}$$

since $ctx = 1$, while `perform (Get_next 2)` can be given the control effect

$$(\forall y.\text{ilist} \to \{z : \text{iopt} \mid \text{isSome}(y) \Rightarrow z \neq \texttt{None}\}) \Rightarrow (q : \text{ilist}) \to \{z : \text{iopt} \mid \text{isCons}(q) \Rightarrow z \neq \texttt{None}\}$$

since $ctx = 2$. Now, the control effect of the argument `body` can be obtained from the composition of these three control effects, which results in

$$(\forall x.(\text{ilist} \to \{z : \text{iopt} \mid \phi_1\})) \Rightarrow (\{z : \text{ilist} \mid \phi_2\} \to \{z : \text{iopt} \mid z \neq \texttt{None}\}) \ .$$

Then, the handling construct is assigned the final answer type of `body`, i.e., $\{z : \text{ilist} \mid \phi_2\} \to \{z : \text{iopt} \mid z \neq \texttt{None}\}$, and finally applying the non-empty initial queue to the handling construct returns a value of type $\{z : \text{iopt} \mid z \neq \texttt{None}\}$ as expected.

C Definitions (other than those shown in the main paper) and Assumptions

C.1 Well-formedness of typing contexts, value types, and computation types

$\boxed{\vdash \Gamma}$ $\boxed{\Gamma \vdash T}$ $\boxed{\Gamma \vdash C}$ $\boxed{\Gamma \vdash \Sigma}$ $\boxed{\Gamma \mid T \vdash S}$

$$\frac{}{\vdash \emptyset}\text{(WE-Empty)} \qquad \frac{\vdash \Gamma \quad x \notin dom(\Gamma) \quad \Gamma \vdash T}{\vdash \Gamma, x : T}\text{(WE-Var)} \qquad \frac{\vdash \Gamma \quad X \notin dom(\Gamma)}{\vdash \Gamma, X : \widetilde{B}}\text{(WE-PVar)}$$

$$\frac{\Gamma, x : B \vdash \phi}{\Gamma \vdash \{x : B \mid \phi\}}\text{(WT-Rfn)} \qquad \frac{\Gamma, x : T \vdash C}{\Gamma \vdash (x : T) \to C}\text{(WT-Fun)}$$

$$\frac{\Gamma \vdash \Sigma \quad \Gamma \vdash T \quad \Gamma \mid T \vdash S}{\Gamma \vdash \Sigma \triangleright T \ / \ S}\text{(WT-Comp)} \qquad \frac{(\Gamma, \widetilde{X_i : \widetilde{B}_i} \vdash F_i)_i}{\Gamma \vdash \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.F_i)_i\}}\text{(WT-Sig)}$$

$$\frac{\vdash \Gamma}{\Gamma \mid T \vdash \square}\text{(WT-Pure)} \qquad \frac{\Gamma, x : T \vdash C_1 \quad \Gamma \vdash C_2}{\Gamma \mid T \vdash (\forall x.C_1) \Rightarrow C_2}\text{(WT-ATM)}$$

C.2 Assumptions on well-formedness judgments of formulas, well-formedness judgments of predicates, and semantic validity judgements of formulas

Assumption 1.

- If $\Gamma \vdash \phi$, then $\vdash \Gamma$.
- If $\vdash \Gamma$, $z \notin dom(\Gamma)$ and $dom(\Gamma, z : B) \supseteq fv(\phi)$, then $\Gamma, z : B \vdash \phi$.
- If $\vdash \Gamma, x : T, \Gamma'$ and $\Gamma, \Gamma' \vdash A : \widetilde{B}$, then $\Gamma, x : T, \Gamma' \vdash A : \widetilde{B}$.
- If $\vdash \Gamma, x : T, \Gamma'$ and $\Gamma, \Gamma' \vdash \phi$, then $\Gamma, x : T, \Gamma' \vdash \phi$.
- If $\Gamma, \Gamma' \models \phi$, then $\Gamma, x : T, \Gamma' \models \phi$.
- If $\Gamma \vdash v : T$ and $\Gamma, x : T, \Gamma' \vdash A : \widetilde{B}$, then $\Gamma, \Gamma'[v/x] \vdash A[v/x] : \widetilde{B}$.
- If $\Gamma \vdash v : T$ and $\Gamma, x : T, \Gamma' \vdash \phi$, then $\Gamma, \Gamma'[v/x] \vdash \phi[v/x]$.
- If $\Gamma \vdash v : T$ and $\Gamma, x : T, \Gamma' \models \phi$, then $\Gamma, \Gamma'[v/x] \models \phi[v/x]$.
- If $\Gamma \vdash A : \widetilde{B}$ and $\Gamma, X : \widetilde{B}, \Gamma' \vdash A' : \widetilde{B'}$, then $\Gamma, \Gamma'[A/X] \vdash A'[A/X] : \widetilde{B'}$.
- If $\Gamma \vdash A : \widetilde{B}$ and $\Gamma, X : \widetilde{B}, \Gamma' \vdash \phi$, then $\Gamma, \Gamma'[A/X] \vdash \phi[A/X]$.
- If $\Gamma \vdash A : \widetilde{B}$ and $\Gamma, X : \widetilde{B}, \Gamma' \models \phi$, then $\Gamma, \Gamma'[A/X] \models \phi[A/X]$.

- If $\Gamma \vdash T_1 <: T_2$, $\vdash \Gamma, x : T_1, \Gamma'$ and $\Gamma, x : T_2, \Gamma' \vdash A : \widetilde{B}$, then $\Gamma, x : T_1, \Gamma' \vdash A : \widetilde{B}$.
- If $\Gamma \vdash T_1 <: T_2$, $\vdash \Gamma, x : T_1, \Gamma'$ and $\Gamma, x : T_2, \Gamma' \vdash \phi$, then $\Gamma, x : T_1, \Gamma' \vdash \phi$.
- If $\Gamma \vdash T_1 <: T_2$ and $\Gamma, x : T_2, \Gamma' \vDash \phi$, then $\Gamma, x : T_1, \Gamma' \vDash \phi$.
- If $x \notin fv(\Gamma', \phi)$ and $\Gamma, x : T_0, \Gamma' \vdash \phi$, then $\Gamma, \Gamma' \vdash \phi$.
- If $\Gamma, x : (y : T_1) \to C_1, \Gamma' \vdash \phi$, then $x \notin fv(\Gamma', \phi)$.
- If $\vDash \phi$ and $\Gamma, \phi, \Gamma' \vDash \phi'$, then $\Gamma, \Gamma' \vDash \phi'$.
- If $\Gamma \vdash \phi$, then $\Gamma \vDash \phi \Rightarrow \phi$.
- If $\Gamma \vDash \phi_1 \Rightarrow \phi_2$ and $\Gamma \vDash \phi_2 \Rightarrow \phi_3$, then $\Gamma \vDash \phi_1 \Rightarrow \phi_3$.
- If $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash \phi$, then $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vDash \phi \implies \phi[y/x]$.
- If $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash \phi$, then $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vDash \phi[y/x] \implies \phi$.

C.3 Assumptions on primitives

Assumption 2.

- $\vdash ty(p)$ for all p.
- If $ty(p) = (x : T) \to C$, then $\zeta(p, v)$ is defined and $\vdash \zeta(p, v) : C[v/x]$ for all v such that $\vdash v : T$.
- If $ty(p) = \{z : \text{bool} \mid \phi\}$, then $p = \textbf{true}$ or $p = \textbf{false}$.

D Proof of Type Safety

D.1 Progress

Lemma 3 (Weakening)**.**

1. *Assume that* $\vdash \Gamma, x : T_0, \Gamma'$.
   - *If* $\Gamma, \Gamma' \vdash T$, *then* $\Gamma, x : T_0, \Gamma' \vdash T$.
   - *If* $\Gamma, \Gamma' \vdash C$, *then* $\Gamma, x : T_0, \Gamma' \vdash C$.
   - *If* $\Gamma, \Gamma' \vdash \Sigma$, *then* $\Gamma, x : T_0, \Gamma' \vdash \Sigma$.
   - *If* $\Gamma, \Gamma' \mid T \vdash S$, *then* $\Gamma, x : T_0, \Gamma' \mid T \vdash S$.
2. *Assume that* $\vdash \Gamma, x : T_0, \Gamma'$.
   - *If* $\Gamma, \Gamma' \vdash v : T$, *then* $\Gamma, x : T_0, \Gamma' \vdash v : T$.
   - *If* $\Gamma, \Gamma' \vdash c : C$, *then* $\Gamma, x : T_0, \Gamma' \vdash c : C$.
3.
   - *If* $\Gamma, \Gamma' \vdash T_1 <: T_2$, *then* $\Gamma, x : T_0, \Gamma' \vdash T_1 <: T_2$.
   - *If* $\Gamma, \Gamma' \vdash C_1 <: C_2$, *then* $\Gamma, x : T_0, \Gamma' \vdash C_1 <: C_2$.
   - *If* $\Gamma, \Gamma' \vdash \Sigma_1 <: \Sigma_2$, *then* $\Gamma, x : T_0, \Gamma' \vdash \Sigma_1 <: \Sigma_2$.
   - *If* $\Gamma, \Gamma' \mid T \vdash S_1 <: S_2$, *then* $\Gamma, x : T_0, \Gamma' \mid T \vdash S_1 <: S_2$.

Proof. By simultaneous induction on the derivations. The cases for (WT-Rfn), (T-Op) and (S-Rfn) use Assumption 1. □

Lemma 4 (Narrowing)**.**

1. *Assume that* $\Gamma \vdash T_1 <: T_2$ *and* $\vdash \Gamma, x : T_1, \Gamma'$.
   - *If* $\Gamma, x : T_2, \Gamma' \vdash T$, *then* $\Gamma, x : T_1, \Gamma' \vdash T$.
   - *If* $\Gamma, x : T_2, \Gamma' \vdash C$, *then* $\Gamma, x : T_1, \Gamma' \vdash C$.
   - *If* $\Gamma, x : T_2, \Gamma' \vdash \Sigma$, *then* $\Gamma, x : T_1, \Gamma' \vdash \Sigma$.
   - *If* $\Gamma, x : T_2, \Gamma' \mid T \vdash S$, *then* $\Gamma, x : T_1, \Gamma' \mid T \vdash S$.

2. *Assume that* $\Gamma \vdash T_1 <: T_2$ *and* $\vdash \Gamma, x : T_1, \Gamma'$.

   - *If* $\Gamma, x : T_2, \Gamma' \vdash v : T$, *then* $\Gamma, x : T_1, \Gamma' \vdash v : T$.
   - *If* $\Gamma, x : T_2, \Gamma' \vdash c : C$, *then* $\Gamma, x : T_1, \Gamma' \vdash c : C$.

3. *Assume that* $\Gamma \vdash T_1 <: T_2$.

   - *If* $\Gamma, x : T_2, \Gamma' \vdash T_1' <: T_2'$, *then* $\Gamma, x : T_1, \Gamma' \vdash T_1' <: T_2'$.
   - *If* $\Gamma, x : T_2, \Gamma' \vdash C_1 <: C_2$, *then* $\Gamma, x : T_1, \Gamma' \vdash C_1 <: C_2$.
   - *If* $\Gamma, x : T_2, \Gamma' \vdash \Sigma_1 <: \Sigma_2$, *then* $\Gamma, x : T_1, \Gamma' \vdash \Sigma_1 <: \Sigma_2$.
   - *If* $\Gamma, x : T_2, \Gamma' \mid T \vdash S_1 <: S_2$, *then* $\Gamma, x : T_1, \Gamma' \mid T \vdash S_1 <: S_2$.

4. *If* $\Gamma \vdash T_1 <: T_2$ *and* $\Gamma \mid T_2 \vdash S_1 <: S_2$, *then* $\Gamma \mid T_1 \vdash S_1 <: S_2$.

Proof. By simultaneous induction on the derivations. The cases for (WT-Rfn), (T-Op) and (S-Rfn) use Assumption 1. □

Lemma 5 (Substitution)**.**

1. *Assume that* $\Gamma \vdash v : T_0$.

   - *If* $\vdash \Gamma, x : T_0, \Gamma'$, *then* $\vdash \Gamma, \Gamma'[v/x]$.
   - *If* $\Gamma, x : T_0, \Gamma' \vdash T$, *then* $\Gamma, \Gamma'[v/x] \vdash T[v/x]$.
   - *If* $\Gamma, x : T_0, \Gamma' \vdash C$, *then* $\Gamma, \Gamma'[v/x] \vdash C[v/x]$.
   - *If* $\Gamma, x : T_0, \Gamma' \vdash \Sigma$, *then* $\Gamma, \Gamma'[v/x] \vdash \Sigma[v/x]$.
   - *If* $\Gamma, x : T_0, \Gamma' \mid T \vdash S$, *then* $\Gamma, \Gamma'[v/x] \mid T[v/x] \vdash S[v/x]$.

2. *Assume that* $\Gamma \vdash v : T_0$.

   - *If* $\Gamma, x : T_0, \Gamma' \vdash v : T$, *then* $\Gamma, \Gamma'[v/x] \vdash v[v/x] : T[v/x]$.
   - *If* $\Gamma, x : T_0, \Gamma' \vdash c : C$, *then* $\Gamma, \Gamma'[v/x] \vdash c[v/x] : C[v/x]$.

3. *Assume that* $\Gamma \vdash v : T_0$.

   - *If* $\Gamma, x : T_0, \Gamma' \vdash T_1 <: T_2$, *then* $\Gamma, \Gamma'[v/x] \vdash T_1[v/x] <: T_2[v/x]$.
   - *If* $\Gamma, x : T_0, \Gamma' \vdash C_1 <: C_2$, *then* $\Gamma, \Gamma'[v/x] \vdash C_1[v/x] <: C_2[v/x]$.
   - *If* $\Gamma, x : T_0, \Gamma' \vdash \Sigma_1 <: \Sigma_2$, *then* $\Gamma, \Gamma'[v/x] \vdash \Sigma_1[v/x] <: \Sigma_2[v/x]$.
   - *If* $\Gamma, x : T_0, \Gamma' \mid T \vdash S_1 <: S_2$, *then* $\Gamma, \Gamma'[v/x] \mid T \vdash S_1[v/x] <: S_2[v/x]$.

Proof. By simultaneous induction on the derivations. The cases for (WT-Rfn), (T-Op) and (S-Rfn) use Assumption 1. □

Lemma 6 (Predicate Substitution)**.**

1. *Assume that* $\Gamma \vdash A : \widetilde{B}$.

   - *If* $\vdash \Gamma, X : \widetilde{B}, \Gamma'$, *then* $\vdash \Gamma, \Gamma'[A/X]$.
   - *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash T$, *then* $\Gamma, \Gamma'[A/X] \vdash T[A/X]$.
   - *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash C$, *then* $\Gamma, \Gamma'[A/X] \vdash C[A/X]$.
   - *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash \Sigma$, *then* $\Gamma, \Gamma'[A/X] \vdash \Sigma[A/X]$.
   - *If* $\Gamma, X : \widetilde{B}, \Gamma' \mid T \vdash S$, *then* $\Gamma, \Gamma'[A/X] \mid T[A/X] \vdash S[A/X]$.

2. *Assume that* $\Gamma \vdash A : \widetilde{B}$.

   - *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash v : T$, *then* $\Gamma, \Gamma'[A/X] \vdash v[A/X] : T[A/X]$.
   - *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash c : C$, *then* $\Gamma, \Gamma'[A/X] \vdash c[A/X] : C[A/X]$.

3. *Assume that* $\Gamma \vdash A : \widetilde{B}$.

   - *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash T_1 <: T_2$, *then* $\Gamma, \Gamma'[A/X] \vdash T_1[A/X] <: T_2[A/X]$.
   - *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash C_1 <: C_2$, *then* $\Gamma, \Gamma'[A/X] \vdash C_1[A/X] <: C_2[A/X]$.

- *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash \Sigma_1 <: \Sigma_2$*, then* $\Gamma, \Gamma'[A/X] \vdash \Sigma_1[A/X] <: \Sigma_2[A/X]$*.*
- *If* $\Gamma, X : \widetilde{B}, \Gamma' \mid T \vdash S_1 <: S_2$*, then* $\Gamma, \Gamma'[A/X] \mid T \vdash S_1[A/X] <: S_2[A/X]$*.*

Proof. By simultaneous induction on the derivations. The cases for (WT-Rfn), (T-Op) and (S-Rfn) use Assumption 1. □

Lemma 7 (Remove unused type bindings)**.**

- *If* $x \notin fv(\Gamma')$ *and* $\vdash \Gamma, x : T_0, \Gamma'$*, then* $\vdash \Gamma, \Gamma'$*.*
- *If* $x \notin fv(\Gamma', T)$ *and* $\Gamma, x : T_0, \Gamma' \vdash T$*, then* $\Gamma, \Gamma' \vdash T$*.*
- *If* $x \notin fv(\Gamma', C)$ *and* $\Gamma, x : T_0, \Gamma' \vdash C$*, then* $\Gamma, \Gamma' \vdash C$*.*
- *If* $x \notin fv(\Gamma', \Sigma)$ *and* $\Gamma, x : T_0, \Gamma' \vdash \Sigma$*, then* $\Gamma, \Gamma' \vdash \Sigma$*.*
- *If* $x \notin fv(\Gamma', T, S)$ *and* $\Gamma, x : T_0, \Gamma' \mid T \vdash S$*, then* $\Gamma, \Gamma' \mid T \vdash S$*.*

Proof. By simultaneous induction on the derivations. The case for (WT-Rfn) uses Assumption 1. □

Lemma 8 (Variables of non-refinement types do not occur in types)**.**

- *If* $\vdash \Gamma, x : (y : T_1) \to C_1, \Gamma'$*, then* $x \notin fv(\Gamma')$*.*
- *If* $\Gamma, x : (y : T_1) \to C_1, \Gamma' \vdash T$*, then* $x \notin fv(\Gamma', T)$*.*
- *If* $\Gamma, x : (y : T_1) \to C_1, \Gamma' \vdash C$*, then* $x \notin fv(\Gamma', C)$*.*
- *If* $\Gamma, x : (y : T_1) \to C_1, \Gamma' \vdash \Sigma$*, then* $x \notin fv(\Gamma', \Sigma)$*.*
- *If* $\Gamma, x : (y : T_1) \to C_1, \Gamma' \mid T \vdash S$*, then* $x \notin fv(\Gamma', T, S)$*.*

Proof. By simultaneous induction on the derivations. The case for (WT-Rfn) uses Assumption 1. □

Lemma 9 (Remove non-refinement type bindings)**.**

- *If* $\vdash \Gamma, x : (y : T_1) \to C_1, \Gamma'$*, then* $\vdash \Gamma, \Gamma'$*.*
- *If* $\Gamma, x : (y : T_1) \to C_1, \Gamma' \vdash T$*, then* $\Gamma, \Gamma' \vdash T$*.*
- *If* $\Gamma, x : (y : T_1) \to C_1, \Gamma' \vdash C$*, then* $\Gamma, \Gamma' \vdash C$*.*
- *If* $\Gamma, x : (y : T_1) \to C_1, \Gamma' \vdash \Sigma$*, then* $\Gamma, \Gamma' \vdash \Sigma$*.*
- *If* $\Gamma, x : (y : T_1) \to C_1, \Gamma' \mid T \vdash S$*, then* $\Gamma, \Gamma' \mid T \vdash S$*.*

Proof. Immediate by Lemma 8 and 7. □

Lemma 10 (Well-formedness of typing contexts from other judgements)**.**

1. *If* $\Gamma \vdash T$*, then* $\vdash \Gamma$*.*
2. *If* $\Gamma \vdash C$*, then* $\vdash \Gamma$*.*
3. *If* $\Gamma \vdash \Sigma$*, then* $\vdash \Gamma$*.*
4. *If* $\Gamma \mid T \vdash S$*, then* $\vdash \Gamma$*.*

Proof. By simultaneous induction on the derivations. □

Lemma 11 (Well-formedness of types from other judgements)**.**

1. *If* $\Gamma \vdash v : T$*, then* $\Gamma \vdash T$*.*
2. *If* $\Gamma \vdash c : C$*, then* $\Gamma \vdash C$*.*

Proof. By simultaneous induction on the derivations.

1. **Case** (T-CVar)**:** We have
   (i) $v = x$,
   (ii) $T = \{z : B \mid z = x\}$,
   (iii) $\vdash \Gamma$, and

(iv) $\Gamma(x) = \{z : B \mid \phi\}$

for some z, x, and B. W.l.o.g., we can assume that $z \notin dom(\Gamma)$. Also, since (iv) implies $x \in dom(\Gamma)$, it holds that $dom(\Gamma, x : B) \supseteq fv(z = x)$. Then, by the Assumption 1, we have $\Gamma, x : B \vdash z = x$. By (WT-Rfn), we have the conclusion.

Case (T-Var)**:** We have

(i) $v = x$,

(ii) $T = \Gamma(x)$,

(iii) $\vdash \Gamma$, and

(iv) $\Gamma(x) \neq \{z : B \mid \phi\}$ for all z, B, and ϕ

for some x. (ii) implies that Γ is of the form $\Gamma_1, x : T, \Gamma_2$ for some Γ_1 and Γ_2. Therefore, by inverting (iii) repeatedly, we have $\Gamma_1 \vdash T$. By Lemma 3 with (iii), we have the conclusion.

Case (T-Prim)**:** We have

(i) $v = p$,

(ii) $T = ty(p)$, and

(iii) $\vdash \Gamma$

for some p. By Assumption 2, we have $\vdash ty(p)$. By Lemma 3 with (iii), we have the conclusion.

Case (T-Fun)**:** We have

(i) $v = \mathbf{rec}(f, x).c$,

(ii) $T = (x : T_0) \to C$, and

(iii) $\Gamma, x : T_0 \vdash c : C$

for some f, x, c, T_0, and C. By the IH of (iii), we have $\Gamma, f : (x : T_0) \to C, x : T_0 \vdash C$. By Lemma 9, we have $\Gamma, x : T_0 \vdash C$. By (WT-Fun), we have the conclusion.

Case (T-VSub)**:** Immediate by inversion.

2. **Case** (T-Ret)**:** We have

(i) $c = \mathbf{return}\ v$,

(ii) $C = \emptyset \triangleright T \ / \ \square$, and

(iii) $\Gamma \vdash v : T$

for some v and T. By the IH of (iii), we have $\Gamma \vdash T$. By Lemma 10, we have $\vdash \Gamma$. Then, we have the conclusion by the following derivation:

$$\dfrac{\dfrac{}{\Gamma \vdash \emptyset} \quad \Gamma \vdash T \quad \dfrac{\vdash \Gamma}{\Gamma \mid T \vdash \square}}{\Gamma \vdash \emptyset \triangleright T \ / \ \square}$$

Case (T-App)**:** We have

(i) $c = v_1\ v_2$,

(ii) $C = C_0[v_2/x]$,

(iii) $\Gamma \vdash v_1 : (x : T_0) \to C_0$, and

(iv) $\Gamma \vdash v_2 : T_0$

for some x, v_1, v_2, T_0 and C_0. By the IH of (iii), we have $\Gamma \vdash (x : T_0) \to C_0$. By inversion, we have $\Gamma, x : T_0 \vdash C_0$. By Lemma 5, we have the conclusion.

Case (T-If)**:** We have

(i) $c = \mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$,

(ii) $\Gamma \vdash v : \{x : \mathrm{bool} \mid \phi\}$,

(iii) $\Gamma, v = \mathbf{true} \vdash c_1 : C$, and

(iv) $\Gamma, v = \mathbf{false} \vdash c_2 : C$

for some x, v, c_1, c_2, and ϕ. By the IH of (iii), we have $\Gamma, v = \mathbf{true} \vdash C$. By Lemma 7, we have the conclusion.

Case (T-CSub)**:** Immediate by inversion.

Case (T-LetP)**:** We have

(i) $c = \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2$,

(ii) $C = \Sigma \triangleright T_2 \;/\; \square$,

(iii) $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \;/\; \square$,

(iv) $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \;/\; \square$, and

(v) $x \notin fv(T_2) \cup fv(\Sigma)$

for some x, c_1, c_2, Σ, T_1, and T_2. By the IHs of (iii) and (iv) respectively, we have

- $\Gamma \vdash \Sigma \triangleright T_1 \;/\; \square$ and
- $\Gamma, x : T_1 \vdash \Sigma \triangleright T_2 \;/\; \square$.

By inversion, we have

(vi) $\Gamma \vdash \Sigma$, and

(vii) $\Gamma, x : T_1 \vdash T_2$.

By Lemma 7 with (v) (vii), we have

(viii) $\Gamma \vdash T_2$.

By Lemma 10 with (vi), we have $\vdash \Gamma$. From this fact and (vi) and (viii), we have the conclusion by the following derivation:

$$\frac{\Gamma \vdash \Sigma \quad \Gamma \vdash T_2 \quad \dfrac{\vdash \Gamma}{\Gamma \mid T_2 \vdash \square}}{\Gamma \vdash \Sigma \triangleright T_2 \;/\; \square}$$

Case (T-LetIp)**:** We have

(i) $c = \textbf{let}\ x = c_1\ \textbf{in}\ c_2$,

(ii) $C = \Sigma \triangleright T_2 \;/\; (\forall z.C_{21}) \Rightarrow C_{12}$,

(iii) $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \;/\; (\forall x.C_0) \Rightarrow C_{12}$,

(iv) $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \;/\; (\forall z.C_{21}) \Rightarrow C_0$, and

(v) $x \notin fv(T_2) \cup fv(\Sigma) \cup (fv(C_{21}) \setminus \{z\})$

for some $x, c_1, c_2, \Sigma, T_1, T_2, C_0, C_{12}$ and C_{21}. By the IHs of (iii) and (iv) respectively, we have

- $\Gamma \vdash \Sigma \triangleright T_1 \;/\; (\forall x.C_0) \Rightarrow C_{12}$ and
- $\Gamma, x : T_1 \vdash \Sigma \triangleright T_2 \;/\; (\forall z.C_{21}) \Rightarrow C_0$.

By inversion, we have

(vi) $\Gamma \vdash \Sigma$,

(vii) $\Gamma \mid T_1 \vdash (\forall x.C_0) \Rightarrow C_{12}$,

(viii) $\Gamma, x : T_1 \vdash T_2$, and

(ix) $\Gamma, x : T_1 \mid T_2 \vdash (\forall z.C_{21}) \Rightarrow C_0$.

By Lemma 7 with (v) (viii), we have

(x) $\Gamma \vdash T_2$.

By inversion with (vii) and (ix) respectively, we have

(xi) $\Gamma, x : T_1 \vdash C_0$,

(xii) $\Gamma \vdash C_{12}$, and

(xiii) $\Gamma, x : T_1, z : T_2 \vdash C_{21}$.

W.l.o.g., we can assume $x \neq z$. Then, (v) implies $x \notin fv(C_{21})$. Therefore, by Lemma 7 with (xiii) and (v), we have $\Gamma, z : T_2 \vdash C_{21}$. From this and (vi), (x), and (xii), we have the conclusion by the following derivation:

$$\frac{\Gamma \vdash \Sigma \quad \Gamma \vdash T_2 \quad \dfrac{\Gamma, z : T_2 \vdash C_{21} \quad \Gamma \vdash C_{12}}{\Gamma \mid T_2 \vdash (\forall z.C_{21}) \Rightarrow C_{12}}}{\Gamma \vdash \Sigma \triangleright T_2 \;/\; (\forall z.C_{21}) \Rightarrow C_{12}}$$

Case (T-Op)**:** We have

(i) $c = \mathsf{op}\ v$,

(ii) $C = \Sigma \triangleright T_2[\widetilde{A/X}][v/x] \;/\; (\forall y.C_1[\widetilde{A/X}][v/x]) \Rightarrow C_2[\widetilde{A/X}][v/x]$,

(iii) $\Sigma \ni \mathsf{op} : \forall \widetilde{X : B}.(x : T_1) \to ((y : T_2) \to C_1) \to C_2$,

(iv) $\Gamma \vdash \Sigma$,

(v) $\Gamma \vdash \widetilde{A : B}$, and

(vi) $\Gamma \vdash v : T_1[\widetilde{A/X}]$

for some $x, y, v, \widetilde{X}, \widetilde{A}, \widetilde{\widetilde{B}}, \Sigma, T_1, T_2, C_1$ and C_2. By inversion of (iv) with (iii), we have

$$\Gamma, \widetilde{X : \widetilde{B}} \vdash (x : T_1) \to ((y : T_2) \to C_1) \to C_2 \ .$$

By more inversion and Lemma 9, we have

- $\Gamma, \widetilde{X : \widetilde{B}}, x : T_1 \vdash T_2$,
- $\Gamma, \widetilde{X : \widetilde{B}}, x : T_1, y : T_2 \vdash C_1$, and
- $\Gamma, \widetilde{X : \widetilde{B}}, x : T_1 \vdash C_2$.

By Lemma 6 with (v) and Lemma 5 with (vi), we have

- $\Gamma \vdash T_2[\widetilde{A/X}][v/x]$,
- $\Gamma, y : T_2[\widetilde{A/X}][v/x] \vdash C_1[\widetilde{A/X}][v/x]$, and
- $\Gamma \vdash C_2[\widetilde{A/X}][v/x]$.

From these and (iv), we have the conclusion by the following derivation:

$$\dfrac{\Gamma \vdash \Sigma \quad \Gamma \vdash T_2[\widetilde{A/X}][v/x] \quad \dfrac{\Gamma, y : T_2[\widetilde{A/X}][v/x] \vdash C_1[\widetilde{A/X}][v/x] \quad \Gamma \vdash C_2[\widetilde{A/X}][v/x]}{\Gamma \mid T_2[\widetilde{A/X}][v/x] \vdash (\forall y.C_1[\widetilde{A/X}][v/x]) \Rightarrow C_2[\widetilde{A/X}][v/x]}}{\Gamma \vdash \Sigma \triangleright T_2[\widetilde{A/X}][v/x] \ / \ (\forall y.C_1[\widetilde{A/X}][v/x]) \Rightarrow C_2[\widetilde{A/X}][v/x]}$$

Case (T-Hndl)**:** We have

(i) $c = \textbf{with } h \textbf{ handle } c_0$,

(ii) $\Gamma \vdash c_0 : \Sigma \triangleright T \ / \ (\forall x_r.C_1) \Rightarrow C$

for some x_r, h, c_0, Σ, T and C_1. We have the conclusion by applying inversion twice to (ii).

□

Lemma 12 (Canonical forms)**.**

1. *If* $\vdash v : (x : T) \to C$, *then (i)* $v = \mathbf{rec}(f, x).c$ *for some* f, c, *or (ii)* $v = p$ *for some* p *and* $\zeta(p, v)$ *is defined for all* v *such that* $\vdash v : T$.

2. *If* $\vdash v : \{x : \text{bool} \mid \phi\}$, *then* $v = \mathbf{true}$ *or* $v = \mathbf{false}$.

Proof. By induction on the derivations.

1. **Case** (T-Fun)**:** Obvious.

   Case (T-Prim)**:** Immediate from Assumption 2.

   Case (T-VSub)**:** By the IH and inversion of the subtyping judgment. The case for (ii) uses Lemma 11.

   Otherwise: Contradictory.

2. **Case** (T-Prim)**:** Immediate from Assumption 2.

   Case (T-VSub)**:** By the IH and inversion of the subtyping judgment.

   Otherwise: Contradictory.

□

Theorem 13 (Progress)**.** *If* $\emptyset \vdash c : \Sigma \triangleright T \ / \ S$, *then either*

- $c = \textbf{return } v$ *for some* v *such that* $\emptyset \vdash v : T$,
- $c = K[\mathsf{op}\ v]$ *for some* K, op *and* v *such that* $\mathsf{op} \in dom(\Sigma)$, *or*
- $c \longrightarrow c'$ *for some* c'.

Proof. By induction on the derivation.

Case (T-Ret) **and** (T-Op)**:** Obvious.

Case (T-CSub)**:** By the IH. Note that $\vdash \Sigma' <: \Sigma$ implies $dom(\Sigma') \supseteq dom(\Sigma)$.

Case (T-APP)**:** We have

(i) $c = v_1\ v_2$,

(ii) $\vdash v_1 : (x : T_1) \to \Sigma \triangleright T \ /\ S$, and

(iii) $\vdash v_2 : T_1$

for some v_1, v_2, x, and T_1. By Lemma 12 with (ii), either one of the following two cases holds.

- $v_1 = \mathbf{rec}(f, x).c_1$ for some f, c_1:
  By (E-APP), we have $(\mathbf{rec}(f, x).c_1)\ v_2 \longrightarrow c_1[v_2/x][(\mathbf{rec}(f, x).c_1)/f]$.
- $v_1 = p$ for some p and $\zeta(p, v)$ is defined for all v such that $\vdash v : T_1$:
  As (iii) holds, $\zeta(p, v_2)$ is defined. Therefore, by (E-PRIM) we have $p\ v_2 \longrightarrow \zeta(p, v_2)$.

Case (T-IF)**:** We have

(i) $c = \mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$,

(ii) $\vdash v : \{x : \text{bool} \mid \phi\}$,

(iii) $v = \mathbf{true} \vdash c_1 : \Sigma \triangleright T \ /\ S$, and

(iv) $v = \mathbf{false} \vdash c_2 : \Sigma \triangleright T \ /\ S$

for some v, c_1, c_2, x, and ϕ. By Lemma 12 with (ii), either one of the following two cases holds.

- $v = \mathbf{true}$: By (E-IFT), we have $\mathbf{if\ true\ then}\ c_1\ \mathbf{else}\ c_2 \longrightarrow c_1$.
- $v = \mathbf{false}$: By (E-IFF), we have $\mathbf{if\ false\ then}\ c_1\ \mathbf{else}\ c_2 \longrightarrow c_2$.

Case (T-LETP)**:** We have

(i) $c = \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2$, and

(ii) $\vdash c_1 : \Sigma \triangleright T_1 \ /\ \Box$

for some x, c_1, c_2, and T_1. By the IH of (ii), either one of the following three cases holds.

- $c_1 = \mathbf{return}\ v_1$ for some v_1:
  By (E-LETRET), we have $\mathbf{let}\ x = \mathbf{return}\ v_1\ \mathbf{in}\ c_2 \longrightarrow c_2[v_1/x]$.
- $c_1 = K_1[\mathsf{op}\ v_1]$ for some K_1, op and v_1 s.t. $\mathsf{op} \in \mathit{dom}(\Sigma)$:
  We have the conclusion with $c = K[\mathsf{op}\ v_1]$ where $K = \mathbf{let}\ x = K_1\ \mathbf{in}\ c_1$.
- $c_1 \longrightarrow c_1'$ for some c_1':
  By (E-LET), we have $\mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 \longrightarrow \mathbf{let}\ x = c_1'\ \mathbf{in}\ c_2$.

Case (T-LETIP)**:** Similar to the case for (T-LETP).

Case (T-HNDL)**:** We have

(i) $c = \mathbf{with}\ h\ \mathbf{handle}\ c_0$,

(ii) $h = \{\mathbf{return}\ x_r \mapsto c_r, (\mathsf{op}_i(x_i, k_i) \mapsto c_i)_i\}$,

(iii) $\Sigma_0 = \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : T_{1i}) \to ((y_i : T_{2i}) \to C_{1i}) \to C_{2i})_i\}$, and

(iv) $\vdash c_0 : \Sigma_0 \triangleright T_0 \ /\ (\forall x_r.C_1) \Rightarrow (\Sigma \triangleright T \ /\ S)$

for some $c_0, x_r, c_r, (\mathsf{op}_i)_i, (x_i)_i, (k_i)_i, (c_i)_i, (\widetilde{X_i})_i, (\widetilde{\widetilde{B}_i})_i, (T_{1i})_i, (T_{2i})_i, (C_{1i})_i, (C_{2i})_i, \Sigma_0, T_0$, and C_1. By the IH of (iv), either one of the following three cases holds.

- $c_0 = \mathbf{return}\ v_0$ for some v_0:
  By (E-HNDLRET), we have $\mathbf{with}\ h\ \mathbf{handle\ return}\ v_0 \longrightarrow c_r[v_0/x_r]$.
- $c_0 = K_0[\mathsf{op}\ v_0]$ for some K_0, op, and v_0 s.t. $\mathsf{op} \in \mathit{dom}(\Sigma_0)$:
  Since $\mathsf{op} \in \mathit{dom}(\Sigma_0) = \{(\mathsf{op}_i)_i\}$, there exists some j such that $1 \le j \le |\mathit{dom}(\Sigma)|$ and $\mathsf{op} = \mathsf{op}_j$. Then, by (E-HNDLOP) we have

$$\mathbf{with}\ h\ \mathbf{handle}\ K_0[\mathsf{op}_j\ v_0] \longrightarrow c_j[v_0/x_j][\lambda y.\mathbf{with}\ h\ \mathbf{handle}\ K_0[\mathbf{return}\ y]/k_j] \ .$$

- $c_0 \longrightarrow c_0'$ for some c_0':
  By (E-HNDL), we have $\mathbf{with}\ h\ \mathbf{handle}\ c_0 \longrightarrow \mathbf{with}\ h\ \mathbf{handle}\ c_0'$.

□

D.2 Subject Reduction

Lemma 14 (Remove tautology)**.**

1. *If* $\vdash \Gamma, \phi, \Gamma'$*, then* $\vdash \Gamma, \Gamma'$*.*
2. - *If* $\Gamma, \phi, \Gamma' \vdash T$*, then* $\Gamma, \Gamma' \vdash T$*.*
   - *If* $\Gamma, \phi, \Gamma' \vdash C$*, then* $\Gamma, \Gamma' \vdash C$*.*
   - *If* $\Gamma, \phi, \Gamma' \vdash \Sigma$*, then* $\Gamma, \Gamma' \vdash \Sigma$*.*
   - *If* $\Gamma, \phi, \Gamma' \mid T \vdash S$*, then* $\Gamma, \Gamma' \mid T \vdash S$*.*
3. *Assume that* $\vDash \phi$*.*
   - *If* $\Gamma, \phi, \Gamma' \vdash v : T$*, then* $\Gamma, \Gamma' \vdash v : T$*.*
   - *If* $\Gamma, \phi, \Gamma' \vdash c : C$*, then* $\Gamma, \Gamma' \vdash c : C$*.*
4. *Assume that* $\vDash \phi$*.*
   - *If* $\Gamma, \phi, \Gamma' \vdash T'_1 <: T'_2$*, then* $\Gamma, \Gamma' \vdash T'_1 <: T'_2$*.*
   - *If* $\Gamma, \phi, \Gamma' \vdash C_1 <: C_2$*, then* $\Gamma, \Gamma' \vdash C_1 <: C_2$*.*
   - *If* $\Gamma, \phi, \Gamma' \vdash \Sigma_1 <: \Sigma_2$*, then* $\Gamma, \Gamma' \vdash \Sigma_1 <: \Sigma_2$*.*
   - *If* $\Gamma, \phi, \Gamma' \mid T \vdash S_1 <: S_2$*, then* $\Gamma, \Gamma' \mid T \vdash S_1 <: S_2$*.*

Proof.

1. Immediate by Lemma 7.
2. Immediate by Lemma 7.
3. By simultaneous induction on the derivations.
4. By simultaneous induction on the derivations. The case for (S-RFN) uses Assumption 1.

□

Lemma 15 (Reflexivity of subtyping)**.**

1. *If* $\Gamma \vdash T$*, then* $\Gamma \vdash T <: T$*.*
2. *If* $\Gamma \vdash C$*, then* $\Gamma \vdash C <: C$*.*
3. *If* $\Gamma \vdash \Sigma$*, then* $\Gamma \vdash \Sigma <: \Sigma$*.*
4. *If* $\Gamma \mid T \vdash S$*, then* $\Gamma \mid T \vdash S <: S$*.*

Proof. By simultaneous induction on the derivations. The case for (WT-RFN) uses Assumption 1. □

Lemma 16 (Transitivity of subtyping)**.**

1. *If* $\Gamma \vdash T_1 <: T_2$ *and* $\Gamma \vdash T_2 <: T_3$*, then* $\Gamma \vdash T_1 <: T_3$*.*
2. *If* $\Gamma \vdash C_1 <: C_2$ *and* $\Gamma \vdash C_2 <: C_3$*, then* $\Gamma \vdash C_1 <: C_3$*.*
3. *If* $\Gamma \vdash \Sigma_1 <: \Sigma_2$ *and* $\Gamma \vdash \Sigma_2 <: \Sigma_3$*, then* $\Gamma \vdash \Sigma_1 <: \Sigma_3$*.*
4. *If* $\Gamma \mid T \vdash S_1 <: S_2$ *and* $\Gamma \mid T \vdash S_2 <: S_3$*, then* $\Gamma \mid T \vdash S_1 <: S_3$*.*

Proof. By simultaneous induction on the structure of T_2, C_2, Σ_2 and S_2.

1. Case analysis on $\Gamma \vdash T_1 <: T_2$.

   Case (S-RFN)**:** By inversion, Assumption 1 and (S-RFN).

   Case (S-FUN)**:** By inversion, the IHs, Lemma 4, and (S-FUN).

2. By inversion of the both derivations, we have

   (i) $\Sigma_1 = \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.F_{1i})_i, (\mathsf{op}'_i : \forall \widetilde{X'_i : \widetilde{B'}_i}.F'_{1i})_i, (\mathsf{op}''_i : \forall \widetilde{X''_i : \widetilde{B''}_i}.F''_{1i})_i\}$,

   (ii) $\Sigma_2 = \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.F_{2i})_i, (\mathsf{op}'_i : \forall \widetilde{X'_i : \widetilde{B'}_i}.F'_{2i})_i\}$,

(iii) $\Sigma_3 = \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.F_{2i})_i\}$,

(iv) $(\Gamma, \widetilde{X_i : \widetilde{B}_i} \vdash F_{1i} <: F_{2i})_i$,

(v) $(\Gamma, \widetilde{X_i : \widetilde{B}_i} \vdash F_{2i} <: F_{3i})_i$, and

(vi) $(\Gamma, \widetilde{X'_i : \widetilde{B'}_i} \vdash F'_{1i} <: F'_{2i})_i$.

By the IH with (iv) and (v), we have $(\Gamma, \widetilde{X_i : \widetilde{B}_i} \vdash F_{1i} <: F_{3i})_i$. By (S-Sig), we have the conclusion.

3. By inversion, the IHs, Lemma 4, and (S-Comp).

4. Case analysis on $\Gamma \vdash S_1 <: S_2$.

Case (S-Pure)**:** Since we have $S_1 = \square = S_2$, we have the conclusion immediately from $\Gamma \vdash S_2 <: S_3$.

Case (S-ATM)**:** We have

(i) $S_1 = (\forall x.C_{11}) \Rightarrow C_{12}$,
(ii) $S_2 = (\forall x.C_{21}) \Rightarrow C_{22}$,
(iii) $\Gamma, x : T \vdash C_{21} <: C_{11}$, and
(iv) $\Gamma \vdash C_{12} <: C_{22}$

for some $x, C_{11}, C_{12}, C_{21}$, and C_{22}. Since (ii), the only rule applicable to $\Gamma \vdash S_2 <: S_3$ is (S-ATM). Therefore, by inversion we have

(v) $S_3 = (\forall x.C_{31}) \Rightarrow C_{32}$,
(vi) $\Gamma, x : T \vdash C_{31} <: C_{21}$, and
(vii) $\Gamma \vdash C_{22} <: C_{32}$

for some C_{31} and C_{32}. By the IHs, we have

- $\Gamma, x : T \vdash C_{31} <: C_{11}$ and
- $\Gamma \vdash C_{12} <: C_{32}$.

We have the conclusion by (S-ATM).

Case (S-Embed)**:** We have

(i) $S_1 = \square$,
(ii) $S_2 = (\forall x.C_{21}) \Rightarrow C_{22}$,
(iii) $\Gamma, x : T \vdash C_{21} <: C_{22}$, and
(iv) $x \notin fv(C_{22})$

for some x, C_{21}, and C_{22}. Since (ii), the only rule applicable to $\Gamma \vdash S_2 <: S_3$ is (S-ATM). Therefore, by inversion we have

(v) $S_3 = (\forall x.C_{31}) \Rightarrow C_{32}$,
(vi) $\Gamma, x : T \vdash C_{31} <: C_{21}$, and
(vii) $\Gamma \vdash C_{22} <: C_{32}$

for some C_{31} and C_{32}. W.l.o.g., we can assume that $x \notin fv(C_{32})$. Then, by Lemma 3 with (vii), we have

(viii) $\Gamma, x : T \vdash C_{22} <: C_{32}$.

By the IHs with (iii), (iii) and (viii), we have $\Gamma, x : T \vdash C_{31} <: C_{32}$. Then we have the conclusion by (S-Embed).

□

Lemma 17 (Subtyping with equal variables)**.**

- *If* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash T$, *then* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash T <: T[y/x]$.
- *If* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash T$, *then* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash T[y/x] <: T$.
- *If* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash C$, *then* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash C <: C[y/x]$.
- *If* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash C$, *then* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash C[y/x] <: C$.
- *If* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash \Sigma$, *then* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash \Sigma <: \Sigma[y/x]$.

- *If* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash \Sigma$*, then* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \vdash \Sigma[y/x] <: \Sigma$ *.*
- *If* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \mid T \vdash S$*, then* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \mid T \vdash S <: S[y/x]$ *.*
- *If* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \mid T \vdash S$*, then* $\Gamma, x : \{z : B \mid z = y\}, \Gamma' \mid T \vdash S[y/x] <: S$ *.*

Proof. By simultaneous induction on the derivations. The case for (WT-Rfn) uses Assumption 1. The cases for (WT-Fun) and (WT-Comp) uses Lemma 4. □

Lemma 18 (Inversion)**.**

1. *If* $\Gamma \vdash p : T$*, then*
   - $\Gamma \vdash ty(p) <: T$*, and*
   - $\Gamma \vdash p : ty(p)$*.*
2. *If* $\Gamma \vdash \mathbf{rec}(f, x).c : (x : T) \to C$*, then there exist some* T_0 *and* C_0 *such that*
   - $\Gamma \vdash \mathbf{rec}(f, x).c : (x : T_0) \to C_0$*,*
   - $\Gamma \vdash (x : T_0) \to C_0 <: (x : T) \to C$*, and*
   - $\Gamma, f : (x : T_0) \to C_0, x : T_0 \vdash c : C_0$*.*
3. *If* $\Gamma \vdash \mathbf{return}\ v : \Sigma \triangleright T \ /\ S$*, then there exist some* T' *such that*
   - $\Gamma \vdash T' <: T$*,*
   - $\Gamma \vdash v : T'$*, and*
   - $\Gamma \mid T' \vdash \square <: S$*.*
4. *If* $\Gamma \vdash \mathsf{op}\ v : \Sigma \triangleright T \ /\ S$*, then there exist some* $\widetilde{X}, \widetilde{\widetilde{B}}, \widetilde{A}, x, y, T_1, T_2, C_1, C_2, C_{01}$*, and* C_{02} *such that*
   - $S = (\forall y.C_{01}) \Rightarrow C_{02}$*,*
   - $\Sigma \ni \mathsf{op} : \forall \widetilde{X : \widetilde{B}}.(x : T_1) \to ((y : T_2) \to C_1) \to C_2$*,*
   - $\Gamma \vdash \widetilde{A : \widetilde{B}}$*,*
   - $\Gamma \vdash v : T_1[\widetilde{A/X}]$*,*
   - $\Gamma \vdash T_2[\widetilde{A/X}][v/x] <: T$*,*
   - $\Gamma, y : T_2[\widetilde{A/X}][v/x] \vdash C_{01} <: C_1[\widetilde{A/X}][v/x]$*, and*
   - $\Gamma \vdash C_2[\widetilde{A/X}][v/x] <: C_{02}$ *.*
5. *If* $\Gamma \vdash \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 : \Sigma \triangleright T \ /\ \square$*, then there exists some* T_1 *such that*
   - $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ /\ \square$*,*
   - $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T \ /\ \square$*, and*
   - $x \notin fv(T) \cup fv(\Sigma)$ *.*
6. *If* $\Gamma \vdash \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 : \Sigma \triangleright T \ /\ (\forall z.C_1) \Rightarrow C_2$*, then there exist some* T_1 *and* C_0 *such that*
   - $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ /\ (\forall x.C_0) \Rightarrow C_2$*,*
   - $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T \ /\ (\forall z.C_1) \Rightarrow C_0$*, and*
   - $x \notin fv(T) \cup fv(\Sigma) \cup (fv(C_1) \setminus \{z\})$ *.*

Proof. By induction on the derivations.

1. Straightforward with Lemma 15 and 16.
2. Straightforward with Lemma 15 and 16.
3. Straightforward with Lemma 15 and 16.
4. **Case** (T-Op)**:** Obvious with Lemma 15.

   Case (T-CSub)**:** We have

   (i) $\Gamma \vdash \mathsf{op}\ v : \Sigma' \triangleright T' \ /\ S'$,

(ii) $\Gamma \vdash \Sigma' \triangleright T' \,/\, S' <: \Sigma \triangleright T \,/\, S$, and

(iii) $\Gamma \vdash \Sigma \triangleright T \,/\, S$

for some Σ', T', and S'. By the IH on (i), we have

(iv) $S' = (\forall y.C'_{01}) \Rightarrow C'_{02}$,

(v) $\Sigma' \ni \mathsf{op} : \forall \widetilde{X : \widetilde{B}}.(x : T'_1) \to ((y : T'_2) \to C'_1) \to C'_2$,

(vi) $\Gamma \vdash \widetilde{A : \widetilde{B}}$,

(vii) $\Gamma \vdash v : T'_1[\widetilde{A/X}]$,

(viii) $\Gamma \vdash T'_2[\widetilde{A/X}][v/x] <: T'$,

(ix) $\Gamma, y : T'_2[\widetilde{A/X}][v/x] \vdash C'_{01} <: C'_1[\widetilde{A/X}][v/x]$, and

(x) $\Gamma \vdash C'_2[\widetilde{A/X}][v/x] <: C'_{02}$

for some $\widetilde{X}, \widetilde{\widetilde{B}}, \widetilde{A}, x, y, T'_1, T'_2, C'_1, C'_2, C'_{01}$, and C'_{02}. By inversion of (ii), we have

(xi) $\Gamma \vdash \Sigma <: \Sigma'$,

(xii) $\Gamma \vdash T' <: T$, and

(xiii) $\Gamma \mid T' \vdash S' <: S$.

By inversion of (xiii) with (iv), we have

(xiv) $S = (\forall y.C_{01}) \Rightarrow C_{02}$,

(xv) $\Gamma, y : T' \vdash C_{01} <: C'_{01}$, and

(xvi) $\Gamma \vdash C'_{02} <: C_{02}$

for some C_{01} and C_{02}. On the other hand, by inversion of (xi) with (v) and Lemma 9, we have

(xvii) $\Sigma \ni \mathsf{op} : \forall \widetilde{X : \widetilde{B}}.(x : T_1) \to ((y : T_2) \to C_1) \to C_2$,

and

- $\Gamma, \widetilde{X : \widetilde{B}} \vdash T'_1 <: T_1$,
- $\Gamma, \widetilde{X : \widetilde{B}}, x : T'_1 \vdash T_2 <: T'_2$,
- $\Gamma, \widetilde{X : \widetilde{B}}, x : T'_1, y : T_2 \vdash C'_1 <: C_1$, and
- $\Gamma, \widetilde{X : \widetilde{B}}, x : T'_1 \vdash C_2 <: C'_2$

for some T_1, T_2, C_1 and C_2. Then, by Lemma 6 with (vi) and Lemma 5 with (vii), we have

(xviii) $\Gamma \vdash T'_1[\widetilde{A/X}] <: T_1[\widetilde{A/X}]$,

(xix) $\Gamma \vdash T_2[\widetilde{A/X}][v/x] <: T'_2[\widetilde{A/X}][v/x]$,

(xx) $\Gamma, y : T_2[\widetilde{A/X}][v/x] \vdash C'_1[\widetilde{A/X}][v/x] <: C_1[\widetilde{A/X}][v/x]$, and

(xxi) $\Gamma \vdash C_2[\widetilde{A/X}][v/x] <: C'_2[\widetilde{A/X}][v/x]$.

By subsumption of (vii) with (xviii),

(xxii) $\Gamma \vdash v : T_1[\widetilde{A/X}]$.

By Lemma 16 with (xix), (viii) and (xii), we have

(xxiii) $\Gamma \vdash T_2[\widetilde{A/X}][v/x] <: T'$ and

(xxiv) $\Gamma \vdash T_2[\widetilde{A/X}][v/x] <: T$.

By Lemma 4 with "(ix) and (xix)" and "(xv) and (xxiii)" respectively, we have

- $\Gamma, y : T_2[\widetilde{A/X}][v/x] \vdash C'_{01} <: C'_1[\widetilde{A/X}][v/x]$ and
- $\Gamma, y : T_2[\widetilde{A/X}][v/x] \vdash C_{01} <: C'_{01}$.

Then, by Lemma 16 with these two and (xx), we have

(xxv) $\Gamma, y : T_2[\widetilde{A/X}][v/x] \vdash C_{01} <: C_1[\widetilde{A/X}][v/x]$.

Also, by Lemma 16 with (xxi), (x) and (xvi), we have

(xxvi) $\Gamma \vdash C_2[\widetilde{A/X}][v/x] <: C_{02}$.

From (xvii), (vi), (xxii), (xxiv), (xxv) and (xxvi), we have the conclusion.

5. **Case (T-LetP):** Obvious.

Case (T-LetIp): Contradictory.

Case (T-CSub): We have

(i) $\Gamma \vdash \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 : \Sigma' \triangleright T' \ / \ S'$,

(ii) $\Gamma \vdash \Sigma' \triangleright T' \ / \ S' <: \Sigma \triangleright T \ / \ \square$, and

(iii) $\Gamma \vdash \Sigma \triangleright T \ / \ \square$

for some Σ', T', and S'. By inversion of (ii), we have

(iv) $S = \square$,

(v) $\Gamma \vdash \Sigma' <: \Sigma$, and

(vi) $\Gamma \vdash T' <: T$.

Then, by the IH of (i), we have

(vii) $\Gamma \vdash c_1 : \Sigma' \triangleright T_1 \ / \ \square$ and

(viii) $\Gamma, x : T_1 \vdash c_2 : \Sigma' \triangleright T' \ / \ \square$

for some T_1. By subsumption of (vii) with (v), we have

(ix) $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ / \ \square$.

By Lemma 10 with (viii), we have

(x) $\vdash \Gamma, x : T_1$.

Then it holds that $x \notin \mathit{dom}(\Gamma)$ and hence from (iii) we have

(xi) $x \notin \mathit{fv}(T) \cup \mathit{fv}(\Sigma)$.

Also, by Lemma 3 with (ii) and (x), we have

- $\Gamma, x : T_1 \vdash \Sigma' \triangleright T' \ / \ \square <: \Sigma \triangleright T \ / \ \square$.

Then by subsumption of (viii) we have

(xii) $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T \ / \ \square$.

Now we have the conclusion from (ix), (xii) and (xi).

6. **Case (T-LetP):** Contradictory.

Case (T-LetIp): Obvious.

Case (T-CSub): We have

(i) $\Gamma \vdash \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2 : \Sigma' \triangleright T' \ / \ S'$,

(ii) $\Gamma \vdash \Sigma' \triangleright T' \ / \ S' <: \Sigma \triangleright T \ / \ (\forall z.C_1) \Rightarrow C_2$, and

(iii) $\Gamma \vdash \Sigma \triangleright T \ / \ (\forall z.C_1) \Rightarrow C_2$

for some Σ', T', and S'. By inversion of (ii), we have

(iv) $\Gamma \vdash \Sigma' <: \Sigma$,

(v) $\Gamma \vdash T' <: T$, and

(vi) $\Gamma \mid T' \vdash S' <: (\forall z.C_1) \Rightarrow C_2$.

Case analysis on the derivation of (vi).

Case (S-Pure): Contradictory.

Case (S-ATM): We have

(vii) $S' = (\forall z.C'_1) \Rightarrow C'_2$,

(viii) $\Gamma, z : T' \vdash C_1 <: C'_1$, and

(ix) $\Gamma \vdash C'_2 <: C_2$

for some C'_1 and C'_2. Then, by the IH of (i), we have

(x) $\Gamma \vdash c_1 : \Sigma' \triangleright T_1 \ / \ (\forall x.C_0) \Rightarrow C'_2$ and

(xi) $\Gamma, x : T_1 \vdash c_2 : \Sigma' \triangleright T' \ / \ (\forall z.C'_1) \Rightarrow C_0$

for some T_1 and C_0. By subsumption of (x) with (iv) and (ix), we have

(xii) $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ / \ (\forall x.C_0) \Rightarrow C_2$.

By Lemma 10 with (xi), we have

(xiii) $\vdash \Gamma, x : T_1$.

Then it holds that $x \notin \mathit{dom}(\Gamma)$ and hence from (iii) we have

(xiv) $x \notin \mathit{fv}(T) \cup \mathit{fv}(\Sigma) \cup (\mathit{fv}(C_1) \setminus \{z\})$.

Also, by Lemma 3 with (iv), (v), (viii) and (xiii), we have

- $\Gamma, x : T_1 \vdash \Sigma' <: \Sigma$,
- $\Gamma, x : T_1 \vdash T' <: T$, and
- $\Gamma, x : T_1, z : T' \vdash C_1 <: C_1'$.

Then by subsumption of (xi) we have

(xv) $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T \ / \ (\forall z.C_1) \Rightarrow C_0$.

Now we have the conclusion from (xii), (xv) and (xiv).

Case (S-Embed)**:** We have

(xvi) $S' = \square$,

(xvii) $\Gamma, z : T \vdash C_1 <: C_2$, and

(xviii) $z \notin fv(C_2)$.

Then, by Lemma 18 with (i), we have

(xix) $\Gamma \vdash c_1 : \Sigma' \triangleright T_1 \ / \ \square$ and

(xx) $\Gamma, x : T_1 \vdash c_2 : \Sigma' \triangleright T' \ / \ \square$

for some T_1. By Lemma 10 with (xx), we have

(xxi) $\vdash \Gamma, x : T_1$.

Then it holds that $x \notin dom(\Gamma)$ and hence from (iii) we have

(xxii) $x \notin fv(T) \cup fv(\Sigma) \cup (fv(C_1) \setminus \{z\})$ and

(xxiii) $x \notin fv(C_2)$.

Also, by inversion of (iii) we have $\Gamma \vdash C_2$, and so we have $\Gamma \vdash C_2 <: C_2$ by Lemma 15. Then, by Lemma 3 with (xxi) we have $\Gamma, x : T_1 \vdash C_2 <: C_2$. And hence, by (S-Embed) with (xxiii) we have

- $\Gamma \mid T_1 \vdash \square <: (\forall x.C_2) \Rightarrow C_2$.

Therefore, by subsumption of (xix) with (iv), we have

(xxiv) $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ / \ (\forall x.C_2) \Rightarrow C_2$.

Moreover, by Lemma 3 with (ii) and (xxi), we have

- $\Gamma, x : T_1 \vdash \Sigma' \triangleright T' \ / \ \square <: \Sigma \triangleright T \ / \ (\forall z.C_1) \Rightarrow C_2$.

Then by subsumption of (xx) we have

(xxv) $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T \ / \ (\forall z.C_1) \Rightarrow C_2$.

Now we have the conclusion from (xxiv), (xxv) and (xxii).

□

Lemma 19 (Inversion with pure evaluation contexts)**.** *If* $\Gamma \vdash K[c] : \Sigma \triangleright T \ / \ (\forall z.C_1) \Rightarrow C_2$*, then there exist some* $y, T_1,$ *and* C_0 *such that*

- $\Gamma \vdash c : \Sigma \triangleright T_1 \ / \ (\forall y.C_0) \Rightarrow C_2$ *and*
- $\Gamma, y : T_1 \vdash K[\textbf{return } y] : \Sigma \triangleright T \ / \ (\forall z.C_1) \Rightarrow C_0$.

Proof. By induction on the structure of K.

Case $K = [\ ]$**:** We have $\Gamma \vdash c : \Sigma \triangleright T \ / \ (\forall z.C_1) \Rightarrow C_2$. By α-renaming, we have

(i) $\Gamma \vdash c : \Sigma \triangleright T \ / \ (\forall y.C_1[y/z]) \Rightarrow C_2$.

Therefore, we have the first half of the conclusion with $T_1 = T$ and $C_0 = C_1[y/z]$.

On the other hand, from (i), it holds that

(ii) $\vdash \Gamma, y : T$

by Lemma 11, Lemma 10, and inversion. We show the second half of the conclusion by case analysis on T.

Case that T is a refinement type $\{z_0 : B \mid \phi\}$: By (T-CVar) and (T-Ret) with (ii), it holds that

(iii) $\Gamma, y : T \vdash \textbf{return } y : \emptyset \triangleright \{z_0 : B \mid z_0 = y\} \ / \ \square$.

Also, we have the following subtyping with Lemma 17:

$$\frac{\overline{\Gamma, y : T, z : \{z_0 : B \mid z_0 = y\} \vdash C_1 <: C_1[y/z]}}{\Gamma, y : T \mid \{z_0 : B \mid z_0 = y\} \vdash \square <: (\forall z.C_1) \Rightarrow C_1[y/z]}$$

Then it holds that

(iv) $\Gamma, y : T \vdash \emptyset \triangleright \{z_0 : B \mid z_0 = y\} \;/\; \Box <: \Sigma \triangleright T \;/\; (\forall z.C_1) \Rightarrow C_1[y/z]$

by subtyping. Therefore, by subsumption with (iii) and (iv), we have the conclusion.

Case that T is not a refinement type: By (T-Var) and (T-Ret) with (ii), it holds that

(v) $\Gamma, y : T \vdash \textbf{return } y : \emptyset \triangleright T \;/\; \Box$.

Also, since T is not a refinement type, by Lemma 8 we have $z \notin C_1$ and so $C_1[y/z] = C_1$. Then, we have the following subtyping with Lemma 15:

$$\frac{\overline{\Gamma, y : T, z : T \vdash C_1 <: C_1[y/z]}}{\Gamma, y : T \mid T \vdash \Box <: (\forall z.C_1) \Rightarrow C_1[y/z]}$$

Then it holds that

(vi) $\Gamma, y : T \vdash \emptyset \triangleright T \;/\; \Box <: \Sigma \triangleright T \;/\; (\forall z.C_1) \Rightarrow C_1[y/z]$

by subtyping. Therefore, by subsumption with (v) and (vi), we have the conclusion.

Case $K = \textbf{let } x = K_1 \textbf{ in } c_2$: We have $\Gamma \vdash \textbf{let } x = K_1[c] \textbf{ in } c_2 : \Sigma \triangleright T \;/\; (\forall z.C_1) \Rightarrow C_2$. By Lemma 18, we have

(i) $\Gamma \vdash K_1[c] : \Sigma \triangleright T' \;/\; (\forall x.C') \Rightarrow C_2$,

(ii) $\Gamma, x : T' \vdash c_2 : \Sigma \triangleright T \;/\; (\forall z.C_1) \Rightarrow C'$, and

(iii) $x \notin fv(T) \cup fv(\Sigma) \cup (fv(C_1) \setminus \{z\})$

for some T' and C'. By the IH of (i), we have

(iv) $\Gamma \vdash c : \Sigma \triangleright T_1 \;/\; (\forall y.C_0) \Rightarrow C_2$ and

(v) $\Gamma, y : T_1 \vdash K_1[\textbf{return } y] : \Sigma \triangleright T' \;/\; (\forall x.C') \Rightarrow C_0$

for some y, T_1 and C_0.

By Lemma 10 with (ii), we have $\vdash \Gamma, x : T'$. By inversion, we have $x \notin dom(\Gamma)$ and $\Gamma \vdash T'$. Also, By Lemma 10 with (v), we have $\vdash \Gamma, y : T_1$. Then, by Lemma 3 we have $\Gamma, y : T_1 \vdash T'$. Moreover, w.l.o.g, we can assume $x \neq y$, and so $x \notin dom(\Gamma) \cup \{y\} = dom(\Gamma, y : T_1)$. Then we have $\vdash \Gamma, y : T_1, x : T'$.

Therefore, by Lemma 3 with (ii), we have

$$\Gamma, y : T_1, x : T' \vdash c_2 : \Sigma \triangleright T \;/\; (\forall z.C_1) \Rightarrow C' \;.$$

Then, by (T-LetIp) with (v) and (iii), we have

$$\Gamma, y : T_1 \vdash \textbf{let } x = K_1[\textbf{return } y] \textbf{ in } c_2 : \Sigma \triangleright T \;/\; (\forall z.C_1) \Rightarrow C_0 \;,$$

that is,

(vi) $\Gamma, y : T_1 \vdash K[\textbf{return } y] : \Sigma \triangleright T \;/\; (\forall z.C_1) \Rightarrow C_0$.

Therefore, from (iv) and (vi) we have the conclusion.

□

Theorem 20 (Subject reduction)**.** *If $\emptyset \vdash c : C$ and $c \longrightarrow c'$, then $\emptyset \vdash c' : C$.*

Proof. By induction on the typing derivation.

Case (T-Ret) and (T-Op): Contradictory because there is no evaluation rule for c.

Case (T-App): We have

(i) $c = v_1\ v_2$,

(ii) $C = C_1[v_2/x]$,

(iii) $\vdash v_1 : (x : T_1) \rightarrow C_1$, and

(iv) $\vdash v_2 : T_1$

for some x, v_1, v_2, T_1 and C_1. Case analysis on the evaluation derivation.

Case (E-App): We have

(v) $v_1 = \textbf{rec}(f, x).c_1$, and

(vi) $c' = c_1[v_2/x][(\mathbf{rec}(f,x).c_1)/f]$

for some f, x and c_1. By Lemma 18 with (iii), we have

(vii) $\vdash v_1 : (x : T_0) \to C_0$,

(viii) $\vdash (x : T_0) \to C_0 <: (x : T_1) \to C_1$, and

(ix) $f : (x : T_0) \to C_0, x : T_0 \vdash c : C_0$

for some T_0 and C_0. By Lemma 10 with (ix), inversion, and Lemma 9, we have $\vdash T_0$. Also, by inversion of (viii), we have $\vdash T_1 <: T_0$. Then, By (T-VSub) with (iv), we have $\vdash v_2 : T_0$. Using this and (vii), we have the conclusion by Lemma 5 with (ix).

Case (E-Prim)**:** We have

(x) $v_1 = p$, and

(xi) $c' = \zeta(p, v_2)$

for some p. By Lemma 18 with (iii), we have

(xii) $\vdash p : ty(p)$, and

(xiii) $\vdash ty(p) <: (x : T_1) \to C_1$.

By inversion of (xiii), we have

(xiv) $ty(p) = (x : T_0) \to C_0$,

(xv) $\vdash T_1 <: T_0$, and

(xvi) $x : T_1 \vdash C_0 <: C_1$

for some T_0 and C_0. By Lemma 10 with (xii) and (xiv) and inversion, we have $\vdash T_0$. Then, by (T-VSub) with (iv) and (xv), we have $\vdash v_2 : T_0$. Therefore, by Assumption 2 with (xiv), we have

(xvii) $\vdash \zeta(p, v_2) : C_0[v_2/x]$.

Also, by Lemma 11 with (iii) and inversion, we have

(xviii) $x : T_1 \vdash C_1$.

Using (iv), by Lemma 5 with (xvi) and (xviii) respectively, we have

- $\vdash C_0[v_2/x] <: C_1[v_2/x]$ and
- $\vdash C_1[v_2/x]$.

Therefore, by (T-CSub) with (xvii), we have the conclusion.

Case (T-If)**:** We have

(i) $c = \mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2$,

(ii) $\vdash v : \{x : \mathrm{bool} \mid \phi\}$,

(iii) $v = \mathbf{true} \vdash c_1 : C$, and

(iv) $v = \mathbf{false} \vdash c_2 : C$

for some x, v, c_1, c_2, and ϕ. Case analysis on the evaluation derivation.

Case (E-IfT)**:** We have

(v) $v = \mathbf{true}$, and

(vi) $c' = c_1$.

We have the conclusion by Lemma 14 with (iii).

Case (E-IfF)**:** Similar.

Case (T-CSub)**:** By the IH and (T-CSub).

Case (T-LetP)**:** We have

(i) $c = \mathbf{let}\ x = c_1\ \mathbf{in}\ c_2$,

(ii) $C = \Sigma \triangleright T_2 \ / \ \square$,

(iii) $\vdash c_1 : \Sigma \triangleright T_1 \ / \ \square$,

(iv) $x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \ / \ \square$, and

(v) $x \notin fv(T_2) \cup fv(\Sigma)$

for some x, c_1, c_2, Σ, T_1 and T_2. Case analysis on the evaluation derivation.

Case (E-Let)**:** By the IH and (T-LetP).

Case (E-LetRet)**:** We have

(vi) $c_1 = \textbf{return}\ v$, and

(vii) $c' = c_2[v/x]$

for some v. By Lemma 18 with (iii), we have

(viii) $\vdash T_0 <: T_1$ and

(ix) $\vdash v : T_0$

for some T_0. By Lemma 11 with (iii) and inversion, we have $\vdash T_1$. Then, by (T-VSub) with (ix) and (viii), we have $\vdash v : T_1$. Therefore, by Lemma 5 with (iv), we have

$$\vdash c_2[v/x] : \Sigma \triangleright T_2 \ / \ \square$$

(Note that since (v), it holds that $\Sigma[v/x] = \Sigma$ and $T_2[v/x] = T_2$.) That is, we have the conclusion.

Case (T-LetIp)**:** We have

(i) $c = \textbf{let}\ x = c_1\ \textbf{in}\ c_2$,

(ii) $C = \Sigma \triangleright T_2 \ / \ (\forall z.C_{21}) \Rightarrow C_{12}$,

(iii) $\vdash c_1 : \Sigma \triangleright T_1 \ / \ (\forall x.C_0) \Rightarrow C_{12}$,

(iv) $x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \ / \ (\forall z.C_{21}) \Rightarrow C_0$, and

(v) $x \notin fv(T_2) \cup fv(\Sigma) \cup (fv(C_{21}) \setminus \{z\})$

for some $x, z, c_1, c_2, \Sigma, T_1, T_2, C_0, C_{12}$ and C_{21}. Case analysis on the evaluation derivation.

Case (E-Let)**:** By the IH and (T-LetIp).

Case (E-LetRet)**:** We have

(vi) $c_1 = \textbf{return}\ v$, and

(vii) $c' = c_2[v/x]$

for some v. By Lemma 18 with (iii), we have

(viii) $\vdash T_0 <: T_1$,

(ix) $\vdash v : T_0$, and

(x) $\mid T_0 \vdash \square <: (\forall x.C_0) \Rightarrow C_{12}$

for some T_0. By Lemma 11 with (iii) and inversion, we have $\vdash T_1$. Then, by (T-VSub) with (ix) and (viii), we have $\vdash v : T_1$. Therefore, by Lemma 5 with (iv), we have

(xi) $\vdash c_2[v/x] : \Sigma \triangleright T_2 \ / \ (\forall z.C_{21}) \Rightarrow C_0[v/x]$.

(Note that since (v), it holds that $\Sigma[v/x] = \Sigma$, $T_2[v/x] = T_2$ and $C_{21}[v/x] = C_{21}$.) By inversion of (x), we have

(xii) $x : T_0 \vdash C_0 <: C_{12}$.

By Lemma 11 with (iii) and inversion, we have $\vdash C_{12}$, which means $x \notin fv(C_{12})$. Therefore, by Lemma 5 with (xii), we have

(xiii) $\vdash C_0[v/x] <: C_{12}$.

On the other hand, by Lemma 11 with (iv) and inversion, we have $x : T_1, z : T_2 \vdash C_{21}$. By Lemma 7 with (v), we have $z : T_2 \vdash C_{21}$. Then, by Lemma 15 we have

(xiv) $z : T_2 \vdash C_{21} <: C_{21}$.

Hence, by (S-ATM) with (xiii) and (xiv), we have $\mid T_2 \vdash (\forall z.C_{21}) \Rightarrow C_0[v/x] <: (\forall z.C_{21}) \Rightarrow C_{12}$. Now we have the conclusion by subsumption of (xi).

Case (T-Hndl)**:** We have

(i) $c = \textbf{with}\ h\ \textbf{handle}\ c_0$,

(ii) $h = \{\textbf{return}\ x_r \mapsto c_r, (\mathsf{op}_i(x_i, k_i) \mapsto c_i)_i\}$,

(iii) $\vdash c_0 : \Sigma_0 \triangleright T_0 \ / \ (\forall x_r.C_1) \Rightarrow C$,

(iv) $x_r : T_0 \vdash c_r : C_1$,

(v) $\left(\widetilde{X_i : \widetilde{B}_i}, x_i : T_{i1}, k_i : (y_i : T_{i2}) \to C_{i1} \vdash c_i : C_{i2}\right)_i$, and

(vi) $\Sigma_0 = \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : T_{i1}) \to ((y_i : T_{i2}) \to C_{i1}) \to C_{2i})_i\}$

Case analysis on the evaluation derivation.

Case (E-HNDL)**:** By the IH and (T-HNDL).

Case (E-HNDLRET)**:** We have

(vii) $c_0 = \textbf{return}\ v$ and

(viii) $c' = c_r[v/x_r]$

for some v. By Lemma 18 with (iii), we have

(ix) $\vdash T_0' <: T_0$,

(x) $\vdash v : T_0'$, and

(xi) $\mid T_0' \vdash \Box <: (\forall x_r.C_1) \Rightarrow C$

for some T_0'. By inversion of (xi), we have

(xii) $x_r : T_0' \vdash C_1 <: C$ and

(xiii) $x_r \notin fv(C)$.

By Lemma 4 with (iv) and (ix), we have

(xiv) $x_r : T_0' \vdash c_r : C_1$.

By Lemma 5 with (x) applied to (xii) and (xiv), we have

(xv) $\vdash C_1[v/x_r] <: C$ and

(xvi) $\vdash c_r[v/x_r] : C_1[v/x_r]$

respectively. (Note that $C[v/x_r] = C$ since (xiii).) By Lemma 11 with (iii) and inversion, we have $\vdash C$. From this and (xv) and (xvi), we have the conclusion by (T-CSUB).

Case (E-HNDLOP)**:** We have

(xvii) $c_0 = K[\mathsf{op}_i\ v]$ and

(xviii) $c' = c_i[v/x_i][(\lambda y.\textbf{with}\ h\ \textbf{handle}\ K[\textbf{return}\ y])/k_i]$

for some K and v. W.l.o.g., we can assume that y is disjoint from any other existing variables. By Lemma 19 with (iii), we have

(xix) $\vdash \mathsf{op}\ v : \Sigma_0 \triangleright T_1\ /\ (\forall y.C_0) \Rightarrow C$ and

(xx) $y : T_1 \vdash K[\textbf{return}\ y] : \Sigma_0 \triangleright T_0\ /\ (\forall x_r.C_1) \Rightarrow C_0$

for some y, T_1 and C_0. By Lemma 18 with (xix), we have

(xxi) $\Sigma_0 \ni \mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : T_{i1}) \to ((y : T_{i2}) \to C_{i1}) \to C_{i2}$,

(xxii) $\vdash \widetilde{A : \widetilde{B}_i}$,

(xxiii) $\vdash v : T_{i1}[\widetilde{A/X_i}]$,

(xxiv) $\vdash T_{i2}[\widetilde{A/X_i}][v/x_i] <: T_1$,

(xxv) $y : T_{i2}[\widetilde{A/X_i}][v/x_i] \vdash C_0 <: C_{i1}[\widetilde{A/X_i}][v/x_i]$, and

(xxvi) $\vdash C_{i2}[\widetilde{A/X_i}][v/x_i] <: C$

for some $\widetilde{A}$. Note that since (vi) holds, it holds that $y = y_i$ and we use $\widetilde{X_i}, \widetilde{\widetilde{B}_i}, x_i, T_{i1}, T_{i2}, C_{i1}$, and C_{i2} here instead of introducing new ones.

Also, by Lemma 4 with (xx) and (xxiv), we have

$$y : T_{i2}[\widetilde{A/X_i}][v/x_i] \vdash K[\textbf{return}\ y] : \Sigma_0 \triangleright T_0\ /\ (\forall x_r.C_1) \Rightarrow C_0\ .$$

Then, by subsumption with (xxv), we have

(xxvii) $y : T_{i2}[\widetilde{A/X_i}][v/x_i] \vdash K[\textbf{return}\ y] : \Sigma_0 \triangleright T_0\ /\ (\forall x_r.C_1) \Rightarrow C_{i1}[\widetilde{A/X_i}][v/x_i]$.

On the other hand, by Lemma 10 with (xxvii) we have $\vdash y : T_{i2}[\widetilde{A/X_i}][v/x_i]$, and hence by Lemma 3 with (iv) and (v), we have

(xxviii) $y : T_{i2}[\widetilde{A/X_i}][v/x_i], x_r : T_0 \vdash c_r : C_1$ and

(xxix) $\left(y : T_{i2}[\widetilde{A/X_i}][v/x_i], \widetilde{X_i : \widetilde{B}_i}, x_i : T_{i1}, k_i : (y_i : T_{i2}) \to C_{i1} \vdash c_i : C_{i2}\right)_i$.

Therefore, by (T-HNDL) with (ii), (vi), (xxvii), (xxviii), and (xxix), we have

$$y : T_{i2}[\widetilde{A/X_i}][v/x_i] \vdash \textbf{with } h \textbf{ handle } K[\textbf{return } y] : C_{i1}[\widetilde{A/X_i}][v/x_i] \ .$$

Then by (T-FUN) we have

$$\text{(xxx)} \ \vdash \lambda y.\textbf{with } h \textbf{ handle } K[\textbf{return } y] : (y : y : T_{i2}[\widetilde{A/X_i}][v/x_i]) \to C_{i1}[\widetilde{A/X_i}][v/x_i] \ .$$

Now, by Lemma 6 with (xxii) applied to (v), we have

$$x_i : T_{i1}[\widetilde{A/X_i}], k_i : (y_i : T_{i2}[\widetilde{A/X_i}]) \to C_{i1}[\widetilde{A/X_i}] \vdash c_i : C_{i2}[\widetilde{A/X_i}] \ .$$

By applying 5 twice with (xxiii) and (xxx) in a row, we have

$$\vdash c_i[v/x_i][(\lambda y.\textbf{with } h \textbf{ handle } K[\textbf{return } y])/k_i] : C_{i2}[\widetilde{A/X_i}][v/x_i] \ .$$

Note that $C_{i2}[\widetilde{A/X_i}][v/x_i][(\lambda y.\textbf{with } h \textbf{ handle } K[\textbf{return } y])/k_i] = C_{i2}[\widetilde{A/X_i}][v/x_i]$ since $k_i \notin fv(C_{i2}[\widetilde{A/X_i}][v/x_i])$ by Lemma 8. Now we have the conclusion by subsumption with (xxvi).

□

D.3 Type Safety

Theorem 21 (Type safety). *If $\emptyset \vdash c : \Sigma \triangleright T \ / \ S$ and $c \longrightarrow^* c'$, then either:*

- $c' = \textbf{return } v$ *for some v such that $\emptyset \vdash v : T$,*
- $c' = K[\mathsf{op}\ v]$ *for some K, op and v such that $\mathsf{op} \in dom(\Sigma)$, or*
- $c' \longrightarrow c''$ *for some c'' such that $\emptyset \vdash c'' : \Sigma \triangleright T \ / \ S$.*

Proof. By induction on the length of $\longrightarrow^*$ with Theorem 13 and Theorem 20. □

E Definitions for the CPS transformation

E.1 Evaluation rules for the target language of the CPS transformation

$$\text{evaluation context} \quad E ::= [\,] \mid E\ v \mid E\ \widetilde{A} \mid E\ \tau$$

$\boxed{c \longrightarrow c'}$

$$\frac{c \longrightarrow c'}{E[c] \longrightarrow E[c']}\text{(EC-CTX)} \qquad \frac{}{\textbf{if true then } c_1 \textbf{ else } c_2 \longrightarrow c_1}\text{(EC-IFT)} \qquad \frac{}{\textbf{if false then } c_1 \textbf{ else } c_2 \longrightarrow c_2}\text{(EC-IFF)}$$

$$\frac{}{(\textbf{rec}(f : \tau_1, x : \tau_2).c)\ v \longrightarrow c[v/x][(\textbf{rec}(f : \tau_1, x : \tau_2).c)/f]}\text{(EC-APP)}$$

$$\frac{}{p\ v \longrightarrow \zeta_{cps}(p, v)}\text{(EC-PRIM)} \qquad \frac{}{(\Lambda \widetilde{X : \widetilde{B}}.c)\ \widetilde{A} \longrightarrow c[\widetilde{A/X}]}\text{(EC-PAPP)}$$

$$\frac{}{\{(\mathsf{op}_i = v_i)_i\}\#\mathsf{op}_i \longrightarrow v_i}\text{(EC-PROJ)} \qquad \frac{}{(\Lambda\alpha.c)\ \tau \longrightarrow c[\tau/\alpha]}\text{(EC-TAPP)} \qquad \frac{}{(c : \tau) \longrightarrow c}\text{(EC-ACSR)}$$

E.2 Syntax of typing contexts of the target language of the CPS transformation

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, X : \widetilde{B} \mid \Gamma, \alpha$$

E.3 Well-formedness rules of the target language of the CPS transformation

$\boxed{\vdash \Gamma}$ $\boxed{\Gamma \vdash \tau}$

$$\frac{}{\vdash \emptyset}\,(\textsc{WEc-Empty}) \qquad \frac{\vdash \Gamma \quad x \notin \mathit{dom}(\Gamma) \quad \Gamma \vdash \tau}{\vdash \Gamma, x : \tau}\,(\textsc{WEc-Var})$$

$$\frac{\vdash \Gamma \quad X \notin \mathit{dom}(\Gamma)}{\vdash \Gamma, X : \widetilde{B}}\,(\textsc{WEc-PVar}) \qquad \frac{\vdash \Gamma \quad \alpha \notin \mathit{dom}(\Gamma)}{\vdash \Gamma, \alpha}\,(\textsc{WEc-TVar})$$

$$\frac{\Gamma, x : B \vdash \phi}{\Gamma \vdash \{x : B \mid \phi\}}\,(\textsc{WTc-Rfn}) \qquad \frac{\Gamma, x : \tau_1 \vdash \tau_2}{\Gamma \vdash (x : \tau_1) \to \tau_2}\,(\textsc{WTc-Fun}) \qquad \frac{\Gamma, \widetilde{X : \widetilde{B}} \vdash \tau}{\Gamma \vdash \forall \widetilde{X : \widetilde{B}}.\tau}\,(\textsc{WTc-PPoly})$$

$$\frac{(\Gamma \vdash \tau_i)_i}{\Gamma \vdash \{(\mathsf{op}_i : \tau_i)_i\}}\,(\textsc{WTc-Rcd}) \qquad \frac{\alpha \in \Gamma}{\Gamma \vdash \alpha}\,(\textsc{WTc-TVar}) \qquad \frac{\Gamma, \alpha \vdash \tau}{\Gamma \vdash \forall \alpha.\tau}\,(\textsc{WTc-TPoly})$$

E.4 Typing rules of the target language of the CPS transformation

$\boxed{\Gamma \vdash c : \tau}$

$$\frac{\vdash \Gamma \quad \Gamma(x) = \{y : B \mid \phi\}}{\Gamma \vdash x : \{y : B \mid x = y\}}\,(\textsc{Tc-CVar}) \qquad \frac{\vdash \Gamma \quad \forall y, B, \phi. \Gamma(x) \neq \{y : B \mid \phi\}}{\Gamma \vdash x : \Gamma(x)}\,(\textsc{Tc-Var}) \qquad \frac{\vdash \Gamma}{\Gamma \vdash p : \mathit{ty}_{\mathit{cps}}(p)}\,(\textsc{Tc-Prim})$$

$$\frac{\Gamma, f : (x : \tau_1) \to \tau_2, x : \tau_1 \vdash c : \tau_2}{\Gamma \vdash \mathbf{rec}(f : (x : \tau_1) \to \tau_2, x : \tau_1).c : (x : \tau_1) \to \tau_2}\,(\textsc{Tc-Fun}) \qquad \frac{\Gamma \vdash c : (x : \tau_1) \to \tau_2 \quad \Gamma \vdash v : \tau_1}{\Gamma \vdash c\ v : \tau_2[v/x]}\,(\textsc{Tc-App})$$

$$\frac{\Gamma, \alpha \vdash c : \tau}{\Gamma \vdash \Lambda\alpha.c : \forall\alpha.\tau}\,(\textsc{Tc-TAbs}) \qquad \frac{\Gamma \vdash c : \forall\alpha.\tau' \quad \Gamma \vdash \tau}{\Gamma \vdash c\ \tau : \tau'[\tau/\alpha]}\,(\textsc{Tc-TApp})$$

$$\frac{\Gamma, \widetilde{X : \widetilde{B}} \vdash c : \tau}{\Gamma \vdash \Lambda \widetilde{X : \widetilde{B}}.c : \forall \widetilde{X : \widetilde{B}}.\tau}\,(\textsc{Tc-PAbs}) \qquad \frac{\Gamma \vdash c : \forall \widetilde{X : \widetilde{B}}.\tau \quad \widetilde{\Gamma \vdash A : \widetilde{B}}}{\Gamma \vdash c\ \widetilde{A} : \tau[\widetilde{A/X}]}\,(\textsc{Tc-PApp})$$

$$\frac{(\Gamma \vdash v_i : \tau_i)_i}{\Gamma \vdash \{(\mathsf{op}_i = v_i)_i\} : \{(\mathsf{op}_i : \tau_i)_i\}}\,(\textsc{Tc-Rcd}) \qquad \frac{\Gamma \vdash v : \{(\mathsf{op}_i : \tau_i)_i\}}{\Gamma \vdash v\#\mathsf{op}_i : \tau_i}\,(\textsc{Tc-Proj})$$

$$\frac{\Gamma \vdash v : \{x : \mathrm{bool} \mid \phi\} \quad \Gamma, v = \mathbf{true} \vdash c_1 : \tau \quad \Gamma, v = \mathbf{false} \vdash c_2 : \tau}{\Gamma \vdash \mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau}\,(\textsc{Tc-If})$$

$$\frac{\Gamma \vdash c : \tau' \quad \Gamma \vdash \tau' <: \tau \quad \Gamma \vdash \tau}{\Gamma \vdash (c : \tau) : \tau}\,(\textsc{Tc-Ascr}) \qquad \frac{\Gamma \vdash c : \tau_1 \quad \Gamma \vdash \tau_1 <: \tau_2 \quad \Gamma \vdash \tau_2}{\Gamma \vdash c : \tau_2}\,(\textsc{Tc-Sub})$$

E.5 Subtyping rules of the target language of the CPS transformation

$\boxed{\Gamma \vdash \tau_1 <: \tau_2}$

$$\frac{\Gamma, x : B \vDash \phi_1 \implies \phi_2}{\Gamma \vdash \{x : B \mid \phi_1\} <: \{x : B \mid \phi_2\}}\,(\textsc{Sc-Rfn}) \qquad \frac{\Gamma \vdash \tau_{21} <: \tau_{11} \quad \Gamma, x : \tau_{21} \vdash \tau_{12} <: \tau_{22}}{\Gamma \vdash (x : \tau_{11}) \to \tau_{12} <: (x : \tau_{21}) \to \tau_{22}}\,(\textsc{Sc-Fun})$$

$$\frac{\Gamma, \widetilde{X : \widetilde{B}} \vdash \tau_1 <: \tau_2}{\Gamma \vdash \forall \widetilde{X : \widetilde{B}}.\tau_1 <: \forall \widetilde{X : \widetilde{B}}.\tau_2}\,(\textsc{Sc-PPoly}) \qquad \frac{(\Gamma \vdash \tau_{1i} <: \tau_{2i})_i}{\Gamma \vdash \{(\mathsf{op}_i : \tau_{1i})_i, (\mathsf{op}'_i : \tau'_i)_i\} <: \{(\mathsf{op}_i : \tau_{2i})_i\}}\,(\textsc{Sc-Rcd})$$

$$\frac{\alpha \in \Gamma}{\Gamma \vdash \alpha <: \alpha}\,(\textsc{Sc-TVar}) \qquad \frac{\Gamma, \beta \vdash \tau_1[\tau/\alpha] <: \tau_2 \quad \Gamma, \beta \vdash \tau \quad \beta \notin \mathit{fv}(\forall\alpha.\tau_1)}{\Gamma \vdash \forall\alpha.\tau_1 <: \forall\beta.\tau_2}\,(\textsc{Sc-Poly})$$

E.6 CPS transformation of expressions

$$\begin{aligned}
\llbracket x \rrbracket &\stackrel{\mathrm{def}}{=} x \\
\llbracket p \rrbracket &\stackrel{\mathrm{def}}{=} \mathit{cps}(p) \\
\llbracket \mathbf{rec}(f^{(x:T_1)\to C_1}, x^{T_2}).c \rrbracket &\stackrel{\mathrm{def}}{=} \mathbf{rec}(f : \llbracket (x : T_1) \to C_1 \rrbracket, x : \llbracket T_2 \rrbracket).\llbracket c \rrbracket \\
\llbracket \mathbf{return}\ v^T \rrbracket &\stackrel{\mathrm{def}}{=} \overline{\Lambda}\alpha.\overline{\lambda}h : \{\}.\overline{\lambda}k : \llbracket T \rrbracket \to \alpha.k\ \llbracket v \rrbracket
\end{aligned}$$

$$
\begin{aligned}
[\![\textbf{let } x = c_1^{\Sigma \triangleright T_1 / \square} \textbf{ in } c_2^{\Sigma \triangleright T_2 / \square}]\!] &\stackrel{\text{def}}{=} \overline{\Lambda}\alpha.\overline{\lambda}h : [\![\Sigma]\!].\overline{\lambda}k : [\![T_2]\!] \to \alpha.[\![c_1]\!] \mathbin{\overline{@}} \alpha \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x : [\![T_1]\!].[\![c_2]\!] \mathbin{\overline{@}} \alpha \mathbin{\overline{@}} h \mathbin{\overline{@}} k) \\
[\![\textbf{let } x = c_1^{\Sigma \triangleright T_1 / (\forall x.C_1) \Rightarrow C_2} \textbf{ in } c_2^{\Sigma \triangleright T_2 / (\forall z.C_0) \Rightarrow C_1}]\!] &\stackrel{\text{def}}{=} \\
\overline{\Lambda}\alpha.\overline{\lambda}h : [\![\Sigma]\!].\overline{\lambda}k : (z : [\![T_2]\!]) \to [\![C_0]\!].[\![c_1]\!] \mathbin{\overline{@}} [\![C_2]\!] \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x : [\![T_1]\!].[\![c_2]\!] \mathbin{\overline{@}} [\![C_1]\!] \mathbin{\overline{@}} h \mathbin{\overline{@}} k) \\
[\![v_1 \ v_2]\!] &\stackrel{\text{def}}{=} [\![v_1]\!] \ [\![v_2]\!] \\
[\![(\textbf{if } v \textbf{ then } c_1 \textbf{ else } c_2)^C]\!] &\stackrel{\text{def}}{=} (\textbf{if } [\![v]\!] \textbf{ then } [\![c_1]\!] \textbf{ else } [\![c_2]\!] : [\![C]\!]) \\
[\![(\mathsf{op}^{\widetilde{A}} \ v)^{\Sigma \triangleright T / (\forall y.C_1) \Rightarrow C_2}]\!] &\stackrel{\text{def}}{=} \overline{\Lambda}\alpha.\overline{\lambda}h : [\![\Sigma]\!].\overline{\lambda}k : (y : [\![T]\!] \to [\![C_1]\!]).h\#\mathsf{op} \ \widetilde{A} \ [\![v]\!] \ (\lambda y' : [\![T]\!].k \ y') \\
[\![(\textbf{with } h \textbf{ handle } c)^C]\!] &\stackrel{\text{def}}{=} [\![c]\!] \mathbin{\overline{@}} [\![C]\!] \mathbin{\overline{@}} [\![h^{ops}]\!] \mathbin{\overline{@}} [\![h^{ret}]\!] \\
\text{where} &\begin{cases} h = \{\textbf{return } x_r^{T_r} \mapsto c_r, (\mathsf{op}_i^{\widetilde{X_i : \widetilde{B}_i}}(x_i^{T_{x_i}}, k_i^{T_{k_i}}) \mapsto c_i)_i\} \\ [\![h^{ops}]\!] \stackrel{\text{def}}{=} \{(\mathsf{op}_i = \Lambda \widetilde{X_i : \widetilde{B}_i}.\lambda x_i : [\![T_{x_i}]\!].\lambda k_i : [\![T_{k_i}]\!].[\![c_i]\!])_i\} \\ [\![h^{ret}]\!] \stackrel{\text{def}}{=} \lambda x_r : [\![T_r]\!].[\![c_r]\!] \end{cases}
\end{aligned}
$$

E.7 CPS transformation of types and typing contexts

$$
\begin{aligned}
[\![\{x : B \mid \phi\}]\!] &\stackrel{\text{def}}{=} \{x : B \mid \phi\} \\
[\![(x : T) \to C]\!] &\stackrel{\text{def}}{=} (x : [\![T]\!]) \to [\![C]\!] \\
[\![\Sigma \triangleright T \ / \ (\forall x.C_1) \Rightarrow C_2]\!] &\stackrel{\text{def}}{=} \forall \_.[\![\Sigma]\!] \to ((x : [\![T]\!]) \to [\![C_1]\!]) \to [\![C_2]\!] \\
[\![\Sigma \triangleright T \ / \ \square]\!] &\stackrel{\text{def}}{=} \forall \alpha.[\![\Sigma]\!] \to ([\![T]\!] \to \alpha) \to \alpha \\
[\![\{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.F_i)_i\}]\!] &\stackrel{\text{def}}{=} \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.[\![F_i]\!]^{\mathcal{F}})_i\} \\
[\![(x : T_1) \to ((y : T_2) \to C_1) \to C_2]\!]^{\mathcal{F}} &\stackrel{\text{def}}{=} (x : [\![T_1]\!]) \to [\![((y : T_2) \to C_1)]\!] \to [\![C_2]\!]
\end{aligned}
$$

$$
\begin{aligned}
[\![\emptyset]\!] &\stackrel{\text{def}}{=} \emptyset \\
[\![\Gamma, x : T]\!] &\stackrel{\text{def}}{=} [\![\Gamma]\!], x : [\![T]\!] \\
[\![\Gamma, X : \widetilde{B}]\!] &\stackrel{\text{def}}{=} [\![\Gamma]\!], X : \widetilde{B}
\end{aligned}
$$

F Proof of dynamic semantics preservation of the CPS transformation

Regarding dynamic semantics, we identify values and computations modulo types and predicates since they are irrelevant to the dynamic semantics. That is, the following equations hold, for example:

$$
\begin{aligned}
\lambda x : \tau_1.c &= \lambda x : \tau_2.c \\
c[\tau/\alpha] &= c \\
c[A/X] &= c
\end{aligned}
$$

Also, We often omit type annotations when they are unnecessary.

Moreover, We also identify values and computations modulo β equivalence of the (static) meta language (this is admissible because the meta language is pure). Formally, we define a relation $\equiv_\beta$ as the smallest congruence relation over expressions in the target language that satisfies the following equations:

$$
\begin{aligned}
(\overline{\lambda} x : \tau.c) \mathbin{\overline{@}} v &\equiv_\beta c[v/x] \\
(\overline{\Lambda} \alpha.c) \mathbin{\overline{@}} \tau &\equiv_\beta c[\tau/\alpha]
\end{aligned}
$$

and we admit the $\equiv_\beta$-equivalence.

Assumption 22.

- $cps(p) \ [\![v]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \longrightarrow^* [\![\zeta(p, v)]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k$
- If $\zeta(p, v)$ is undefined, then $cps(p) \ [\![v]\!]$ gets stuck.

- $p = \textbf{true} \iff cps(p) = \textbf{true}$
- $p = \textbf{false} \iff cps(p) = \textbf{false}$

Lemma 23 (CPS transformation is homomorphic for substitution)**.**

- $[\![v[v_0/x]]\!] = [\![v]\!][[\![v_0]\!]/x]$
- $[\![c[v_0/x]]\!] = [\![c]\!][[\![v_0]\!]/x]$

Proof. By simultaneous induction on the structure of v and c. ☐

Lemma 24 (Evaluation with pure evaluation context)**.**

$$[\![K[\mathsf{op}\ v]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \longrightarrow^* v_h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.[\![K[\textbf{return}\ y]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k)\ .$$

Proof. By induction on the structure of K.

Case $K = [\ ]$**:**

$$\begin{aligned}
\text{LHS} &= [\![\mathsf{op}\ v]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.k\ y)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&\longrightarrow^* v_h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.v_k\ y) \\
&\equiv_\beta v_h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.(\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.k\ y) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&= v_h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.[\![\textbf{return}\ y]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&= \text{RHS}
\end{aligned}$$

Case $K = \textbf{let}\ x = K_1\ \textbf{in}\ c_2$**:**

$$\begin{aligned}
\text{LHS} &= [\![\textbf{let}\ x = K_1[\mathsf{op}\ v]\ \textbf{in}\ c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.[\![K_1[\mathsf{op}\ v]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} k)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&\longrightarrow^* [\![K_1[\mathsf{op}\ v]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&\quad \text{(by the IH)} \\
&\longrightarrow^* v_h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.[\![K_1[\textbf{return}\ y]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k)) \\
&\equiv_\beta v_h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.(\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.[\![K_1[\textbf{return}\ y]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} k)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&= v_h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.[\![\textbf{let}\ x = K_1[\textbf{return}\ y]\ \textbf{in}\ c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&= v_h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y.[\![K[\textbf{return}\ y]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&= \text{RHS}
\end{aligned}$$

☐

Lemma 25 (One-step simulation)**.** *If* $c \longrightarrow c'$, *then* $[\![c]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \longrightarrow^* [\![c']\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k$.

Proof. By induction on the derivation of $c \longrightarrow c'$. In the following, we implicitly use Lemma 23 and the equality $c[\tau/\alpha] = c$ and $c[A/X] = c$ (note that we identify computations modulo types and predicates regarding dynamic semantics).

Case (E-Let)**:**

$$\begin{aligned}
\text{LHS} &= [\![\textbf{let}\ x = c_1\ \textbf{in}\ c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.[\![c_1]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} k)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&\longrightarrow^* [\![c_1]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&\quad \text{(by the IH)} \\
&\longrightarrow^* [\![c_1']\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&\equiv_\beta (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.[\![c_1']\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} k)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= [\![\textbf{let}\ x = c_1'\ \textbf{in}\ c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= \text{RHS}
\end{aligned}$$

Case (E-LetRet)**:** First, w.l.o.g., we can assume that $x \notin fv(v_h) \cup fv(v_k)$. Then,

$$\begin{aligned}
\text{LHS} &= [\![\mathbf{let}\ x = \mathbf{return}\ v\ \mathbf{in}\ c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.[\![\mathbf{return}\ v]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} k)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&\longrightarrow^* [\![\mathbf{return}\ v]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.k\ [\![v]\!]) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k) \\
&\longrightarrow^* (\lambda x.[\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k)\ [\![v]\!] \\
&\longrightarrow ([\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k)[[\![v]\!]/x] \\
&= ([\![c_2]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k)[[\![v]\!]/x] \\
&= [\![c_2]\!][[\![v]\!]/x] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= [\![c_2[v/x]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= \text{RHS}
\end{aligned}$$

Case (E-IfT)**:**

$$\begin{aligned}
\text{LHS} &= [\![(\mathbf{if\ true\ then}\ c_1\ \mathbf{else}\ c_2)^C]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= (\mathbf{if\ true\ then}\ [\![c_1]\!]\ \mathbf{else}\ [\![c_2]\!] : [\![C]\!]) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&\longrightarrow^* [\![c_1]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= \text{RHS}
\end{aligned}$$

Case (E-IfF)**:** similar.

Case (E-App)**:**

$$\begin{aligned}
\text{LHS} &= [\![(\mathbf{rec}(f,x).c)\ v]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= (\mathbf{rec}(f,x).[\![c]\!])\ [\![v]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&\longrightarrow [\![c]\!][\mathbf{rec}(f,x).[\![c]\!]/f, [\![v]\!]/x] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= [\![c[\mathbf{rec}(f,x).c/f, v/x]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= \text{RHS}
\end{aligned}$$

Case (E-Prim)**:** By Assumption 22.

Case (E-Hndl)**:**

$$\begin{aligned}
\text{LHS} &= [\![\mathbf{with}\ h\ \mathbf{handle}\ c]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= ([\![c]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} [\![h^{ops}]\!] \mathbin{\overline{@}} [\![h^{ret}]\!]) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&\quad \text{(by the IH)} \\
&= ([\![c']\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} [\![h^{ops}]\!] \mathbin{\overline{@}} [\![h^{ret}]\!]) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= [\![\mathbf{with}\ h\ \mathbf{handle}\ c']\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= \text{RHS}
\end{aligned}$$

Case (E-HndlRet)**:**

$$\begin{aligned}
\text{LHS} &= [\![\mathbf{with}\ h\ \mathbf{handle\ return}\ v]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= ([\![\mathbf{return}\ v]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} [\![h^{ops}]\!] \mathbin{\overline{@}} [\![h^{ret}]\!]) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= (((\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.k\ [\![v]\!])) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} [\![h^{ops}]\!] \mathbin{\overline{@}} [\![h^{ret}]\!]) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&\longrightarrow^* ([\![h^{ret}]\!]\ [\![v]\!]) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= ((\lambda x_r.[\![c_r]\!])\ [\![v]\!]) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= [\![c_r]\!][[\![v]\!]/x_r] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= [\![c_r[v/x_r]]\!] \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\
&= \text{RHS}
\end{aligned}$$

Case (E-HndlOp)**:**

$$\begin{aligned}
\text{LHS} &= [\![\mathbf{with}\ h\ \mathbf{handle}\ K[\mathsf{op}_i\ v]]\!]\ \overline{@}\ \tau\ \overline{@}\ v_h\ \overline{@}\ v_k \\
&= ([\![K[\mathsf{op}_i\ v]]\!]\ \overline{@}\ \tau\ \overline{@}\ [\![h^{ops}]\!]\ \overline{@}\ [\![h^{ret}]\!])\ \overline{@}\ \tau\ \overline{@}\ v_h\ \overline{@}\ v_k \\
&\text{(by Lemma 24)} \\
&\longrightarrow^* [\![h^{ops}]\!]\#\mathsf{op}_i\ \widetilde{A}\ [\![v]\!]\ (\lambda y.[\![K[\mathbf{return}\ y]]\!]\ \overline{@}\ \tau\ \overline{@}\ [\![h^{ops}]\!]\ \overline{@}\ [\![h^{ret}]\!]) \\
&= [\![h^{ops}]\!]\#\mathsf{op}_i\ \widetilde{A}\ [\![v]\!]\ (\lambda y.[\![\mathbf{with}\ h\ \mathbf{handle}\ K[\mathbf{return}\ y]]\!]) \\
&\longrightarrow (\Lambda\widetilde{X_i}.\lambda x_i.\lambda k_i.[\![c_i]\!])\ \widetilde{A}\ [\![v]\!]\ (\lambda y.[\![\mathbf{with}\ h\ \mathbf{handle}\ K[\mathbf{return}\ y]]\!]) \\
&\longrightarrow^* [\![c_i]\!][[\![v]\!]/x_i][\lambda y.[\![\mathbf{with}\ h\ \mathbf{handle}\ K[\mathbf{return}\ y]]\!]/k_i] \\
&= [\![c_i[v/x_i][\lambda y.\mathbf{with}\ h\ \mathbf{handle}\ K[\mathbf{return}\ y]/k_i]]\!] \\
&= \text{RHS}
\end{aligned}$$

□

Theorem 26 (Forward (multi-step) simulation)**.** *If* $c \longrightarrow^* \mathbf{return}\ v$*, then* $[\![c]\!]\ \overline{@}\ \tau\ \overline{@}\ \{\}\ \overline{@}\ (\lambda x : \tau.x) \longrightarrow^+ [\![v]\!]$ *.*

Proof. By applying Lemma 25 repeatedly, we have

$$[\![c]\!]\ \overline{@}\ \tau\ \overline{@}\ \{\}\ \overline{@}\ (\lambda x : \tau.x) \longrightarrow^* [\![\mathbf{return}\ v]\!]\ \overline{@}\ \tau\ \overline{@}\ \{\}\ \overline{@}\ (\lambda x : \tau.x)\ .$$

Then,

$$\begin{aligned}
&[\![\mathbf{return}\ v]\!]\ \overline{@}\ \tau\ \overline{@}\ \{\}\ \overline{@}\ (\lambda x : \tau.x) \\
&= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.k\ [\![v]\!])\ \overline{@}\ \tau\ \overline{@}\ \{\}\ \overline{@}\ (\lambda x : \tau.x) \\
&\longrightarrow^* (\lambda x.x)\ [\![v]\!] \\
&\longrightarrow [\![v]\!]
\end{aligned}$$

and therefore we have the conclusion. □

Definition 27. We define evaluation contexts E as follows:

$$E ::= [\ ] \mid \mathbf{let}\ x = E\ \mathbf{in}\ c \mid \mathbf{with}\ h\ \mathbf{handle}\ E$$

Definition 28. We define a function *bop* as follows:

$$\begin{aligned}
bop([\ ]) &\stackrel{\text{def}}{=} \emptyset \\
bop(\mathbf{let}\ x = E\ \mathbf{in}\ c) &\stackrel{\text{def}}{=} bop(E) \\
bop(\mathbf{with}\ h\ \mathbf{handle}\ E) &\stackrel{\text{def}}{=} dom(h) \cup bop(E)
\end{aligned}$$

That is, $bop(E)$ is a set of operations that are handled by a handler in E.

We say c *is stuck* if c is irreducible and $c \neq \mathbf{return}\ v$. We proceed the proof of the backward simulation following **?**.

Lemma 29 (Preservation of the specific forms of stuck computations)**.**

1. *If* $c = E[\mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]$ *where* v *is not* **true** *nor* **false***, then* $[\![c]\!]\ \overline{@}\ \tau\ \overline{@}\ v_h\ \overline{@}\ v_k$ *gets stuck.*
2. *If* $c = E[v_1\ v_2]$ *where* v_1 *is not* $\mathbf{rec}(f, x).c$ *nor* p *such that* $\zeta(p, v_2)$ *is defined, then* $[\![c]\!]\ \overline{@}\ \tau\ \overline{@}\ v_h\ \overline{@}\ v_k$ *gets stuck.*
3. *Let* h_0 *be a handler. If* $c = E[\mathsf{op}\ v]$ *where* $\mathsf{op} \notin bop(E) \cup dom(h_0)$*, then* $[\![c]\!]\ \overline{@}\ \tau\ \overline{@}\ [\![h_0^{\mathsf{ops}}]\!]\ \overline{@}\ v_k$ *gets stuck.*

Proof.

1. By induction on the structure of E.

   Case $E = [\ ]$**:**

$$\begin{aligned}
[\![c]\!]\ \overline{@}\ \tau\ \overline{@}\ v_h\ \overline{@}\ v_k &= [\![\mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2]\!]\ \overline{@}\ \tau\ \overline{@}\ v_h\ \overline{@}\ v_k \\
&= \mathbf{if}\ [\![v]\!]\ \mathbf{then}\ [\![c_1]\!]\ \mathbf{else}\ [\![c_2]\!]\ \overline{@}\ \tau\ \overline{@}\ v_h\ \overline{@}\ v_k
\end{aligned}$$

   From Assumption 22, $[\![v]\!]$ is neither **true** nor **false**. Therefore, there is no applicable evaluation rule, and hence this computation is stuck.

Case $E = \textbf{let } x = E_1 \textbf{ in } c$**:**

$$\begin{aligned}\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k &= \llbracket \textbf{let } x = E_1[\textbf{if } v \textbf{ then } c_1 \textbf{ else } c_2] \textbf{ in } c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\ &= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.\llbracket E_1[\textbf{if } v \textbf{ then } c_1 \textbf{ else } c_2] \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x.\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} k)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\ &\longrightarrow^* \llbracket E_1[\textbf{if } v \textbf{ then } c_1 \textbf{ else } c_2] \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} (\lambda x.\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k)\end{aligned}$$

By the IH, this computation gets stuck.

Case $E = \textbf{with } h \textbf{ handle } E_1$**:**

$$\begin{aligned}\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k &= \llbracket \textbf{with } h \textbf{ handle } E_1[\textbf{if } v \textbf{ then } c_1 \textbf{ else } c_2] \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k \\ &= (\llbracket E_1[\textbf{if } v \textbf{ then } c_1 \textbf{ else } c_2] \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h^{ops} \rrbracket \mathbin{\overline{@}} \llbracket h^{ret} \rrbracket) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} v_h \mathbin{\overline{@}} v_k\end{aligned}$$

By the IH, this computation gets stuck.

2. Similar to the case 1.

3. By induction on the structure of E.

Case $E = [\,]$**:**

$$\begin{aligned}\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k &= \llbracket \mathsf{op}\ v \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k \\ &= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.h\#\mathsf{op}\ \widetilde{A}\ \llbracket v \rrbracket\ (\lambda y.\llbracket \textbf{return } y \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} k)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k \\ &\longrightarrow^* h_0^{\mathsf{ops}}\#\mathsf{op}\ \widetilde{A}\ \llbracket v \rrbracket\ (\lambda y.\llbracket \textbf{return } y \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k)\end{aligned}$$

Here, h_0^{ops} does not have a field with op since $\mathsf{op} \notin \mathit{dom}(h_0)$. Therefore, there is no applicable evaluation rule, and hence this computation is stuck.

Case $E = \textbf{let } x = E_1 \textbf{ in } c$**:**

$$\begin{aligned}\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k &= \llbracket \textbf{let } x = E_1[\mathsf{op}\ v] \textbf{ in } c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k \\ &= (\overline{\Lambda}\alpha.\overline{\lambda}h.\overline{\lambda}k.\llbracket E_1[\mathsf{op}\ v] \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} (\lambda x.\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} h \mathbin{\overline{@}} k)) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k \\ &\longrightarrow^* \llbracket E_1[\mathsf{op}\ v] \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} (\lambda x.\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k)\end{aligned}$$

Since $\mathsf{op} \notin \mathit{bop}(E) \cup \mathit{dom}(h_0)$ and $\mathit{bop}(E) = \mathit{bop}(\textbf{let } x = E_1 \textbf{ in } c) = \mathit{bop}(E_1)$, it holds that $\mathsf{op} \notin \mathit{bop}(E_1) \cup \mathit{dom}(h_0)$. Then, by the IH, this computation gets stuck.

Case $E = \textbf{with } h \textbf{ handle } E_1$**:**

$$\begin{aligned}\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k &= \llbracket \textbf{with } h \textbf{ handle } E_1[\mathsf{op}\ v] \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k \\ &= (\llbracket E_1[\mathsf{op}\ v] \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h^{ops} \rrbracket \mathbin{\overline{@}} \llbracket h^{ret} \rrbracket) \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \llbracket h_0^{\mathsf{ops}} \rrbracket \mathbin{\overline{@}} v_k\end{aligned}$$

Here, $\mathsf{op} \notin \mathit{bop}(E) = \mathit{bop}(\textbf{with } h \textbf{ handle } E_1) = \mathit{bop}(E_1) \cup \mathit{dom}(h)$. Therefore, by the IH, this computation gets stuck.

□

Lemma 30 (Preservation of stuck computations)**.** *If c is a stuck computation, then $\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \{\} \mathbin{\overline{@}} (\lambda x : \tau.x)$ also gets stuck.*

Proof. A stuck computation c is either:

- $E[\textbf{if } v \textbf{ then } c_1 \textbf{ else } c_2]$ where v is not **true** nor **false**,
- $E[v_1\ v_2]$ where v_1 is not $\textbf{rec}(f, x).c$ nor p such that $\zeta(p, v_2)$ is defined, or
- $E[\mathsf{op}\ v]$ where $\mathsf{op} \notin \mathit{bop}(E)$.

Therefore, it is immediate from Lemma 29. □

Theorem 31 (Backward simulation)**.** *If $\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \{\} \mathbin{\overline{@}} (\lambda x : \tau.x) \longrightarrow^+ v'$, then $c \longrightarrow^* \textbf{return } v$ and $\llbracket v \rrbracket = v'$.*

Proof. We show this theorem by proving its contraposition: If "$c \longrightarrow^* \textbf{return } v$ and $\llbracket v \rrbracket = v'$" does not hold, then $\llbracket c \rrbracket \mathbin{\overline{@}} \tau \mathbin{\overline{@}} \{\} \mathbin{\overline{@}} (\lambda x : \tau.x) \longrightarrow^+ v'$ also does not hold. We can divide the situation into two cases:

Case that $c \longrightarrow^* \textbf{return } v$ does not hold: There are two possibilities where c does not evaluate to a value-return.

Case that c diverges: Since c diverges, for all natural numbers n, there exists a sequence

$$c \longrightarrow c_1 \longrightarrow \cdots \longrightarrow c_n \ .$$

Then, by Lemma 25, we have a sequence

$$[\![c]\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x) \longrightarrow^+ [\![c_1]\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x) \longrightarrow^+ \cdots \longrightarrow^+ [\![c_n]\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x)$$

for all n. The length of the sequence is at least n, and therefore $[\![c]\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x)$ has evaluation sequences of arbitrary length, which means it cannot be evaluated to a value.

Case that $c \longrightarrow^* c'$ and c' is stuck: By applying Lemma 25 repeatedly, we have

$$[\![c]\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x) \longrightarrow^* [\![c']\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x) \ .$$

Also, by Lemma 30, it holds that $[\![c']\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x)$ gets stuck. Therefore, $[\![c]\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x)$ cannot be evaluated to a value.

Case that $c \longrightarrow^*$ return v holds but $[\![v]\!] = v'$ does not: By Theorem 26, we have

$$[\![c]\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x) \longrightarrow^+ [\![v]\!] \ .$$

Then, from the premise $[\![v]\!] \neq v'$ and the fact that the evaluation of the target language is deterministic, it cannot be the case that $[\![c]\!] \,\overline{@}\, \tau \,\overline{@}\, \{\} \,\overline{@}\, (\lambda x : \tau.x) \longrightarrow^+ v'$.

□

Corollary 32 (Simulation)**.** *If $c \longrightarrow^*$ **return** v, then $[\![c]\!] \overline{@} \tau \overline{@} \{\} \overline{@} (\lambda x : \tau.x) \longrightarrow^+ [\![v]\!]$. Also, if $[\![c]\!] \overline{@} \tau \overline{@} \{\} \overline{@} (\lambda x : \tau.x) \longrightarrow^+ v'$, then $c \longrightarrow^*$ **return** v and $[\![v]\!] = v'$.*

Proof. Immediate from Theorem 26 and 31. □

G Proof of type preservation of the CPS transformation

In the following, we consider static expressions and dynamic ones as identical since the distinction is irrelevant to the discussion on the type preservation. In other words, we write $[\![c]\!] \,\overline{@}\, \tau \,\overline{@}\, v_h \,\overline{@}\, v_k$ as $[\![c]\!] \ \tau \ v_h \ v_k$ below, for example.

G.1 Basic properties for the target language of the CPS transformation

Assumption 33.

- If $\vdash \Gamma$ and $dom(\Gamma) \supseteq fv(\phi)$, then $\Gamma \vdash \phi$.
- If $\Gamma \vdash \phi$, then $\vdash \Gamma$.
- If $\Gamma \vdash \phi$, then $\Gamma \models \phi \Rightarrow \phi$.
- If $\Gamma \models \phi_1 \Rightarrow \phi_2$ and $\Gamma \models \phi_2 \Rightarrow \phi_3$, then $\Gamma \models \phi_1 \Rightarrow \phi_3$.
- If $\Gamma \vdash v : \tau$ and $\Gamma, x : \tau, \Gamma' \vdash A : \widetilde{B}$, then $\Gamma, \Gamma'[v/x] \vdash A[v/x] : \widetilde{B}$.
- If $\Gamma \vdash v : \tau$ and $\Gamma, x : \tau, \Gamma' \vdash \phi$, then $\Gamma, \Gamma'[v/x] \vdash \phi[v/x]$.
- If $\Gamma \vdash v : \tau$ and $\Gamma, x : \tau, \Gamma' \models \phi$, then $\Gamma, \Gamma'[v/x] \models \phi[v/x]$.
- If $\Gamma \vdash A : \widetilde{B}$ and $\Gamma, X : \widetilde{B}, \Gamma' \vdash A' : \widetilde{B'}$, then $\Gamma, \Gamma'[A/X] \vdash A'[A/X] : \widetilde{B'}$.
- If $\Gamma \vdash A : \widetilde{B}$ and $\Gamma, X : \widetilde{B}, \Gamma' \vdash \phi$, then $\Gamma, \Gamma'[A/X] \vdash \phi[A/X]$.
- If $\Gamma \vdash A : \widetilde{B}$ and $\Gamma, X : \widetilde{B}, \Gamma' \models \phi$, then $\Gamma, \Gamma'[A/X] \models \phi[A/X]$.
- If $\vdash \Gamma_1, \Gamma_2, \Gamma_3$ and $\Gamma_1, \Gamma_2 \vdash \phi$, then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \phi$.
- If $\vdash \Gamma_1, \Gamma_2, \Gamma_3$ and $\Gamma_1, \Gamma_2 \vdash A : \widetilde{B}$, then $\Gamma_1, \Gamma_2, \Gamma_3 \vdash A : \widetilde{B}$.
- If $\vdash \Gamma_1, \Gamma_2, \Gamma_3$ and $\Gamma_1, \Gamma_2 \models \phi$, then $\Gamma_1, \Gamma_2, \Gamma_3 \models \phi$.
- If $\Gamma \vdash \tau_1 <: \tau_2$, $\vdash \Gamma, x : \tau_1, \Gamma'$ and $\Gamma, x : \tau_2, \Gamma' \vdash A : \widetilde{B}$, then $\Gamma, x : \tau_1, \Gamma' \vdash A : \widetilde{B}$.

- If $\Gamma \vdash \tau_1 <: \tau_2$, $\vdash \Gamma, x : \tau_1, \Gamma'$ and $\Gamma, x : \tau_2, \Gamma' \vdash \phi$, then $\Gamma, x : \tau_1, \Gamma' \vdash \phi$.
- If $\Gamma \vdash \tau_1 <: \tau_2$ and $\Gamma, x : \tau_2, \Gamma' \models \phi$, then $\Gamma, x : \tau_1, \Gamma' \models \phi$.
- If $x \notin \mathit{fv}(\Gamma', \phi)$ and $\Gamma, x : \tau_0, \Gamma' \vdash \phi$, then $\Gamma, \Gamma' \vdash \phi$.
- If $x \notin \mathit{fv}(\Gamma', A)$ and $\Gamma, x : \tau_0, \Gamma' \vdash A : \widetilde{B}$, then $\Gamma, \Gamma' \vdash A : \widetilde{B}$.
- If $\Gamma, x : \tau, \Gamma' \vdash \phi$ and τ is not a refinement type, then $x \notin \mathit{fv}(\Gamma', \phi)$.
- If $\Gamma, x : \tau, \Gamma' \vdash A : \widetilde{B}$ and τ is not a refinement type, then $x \notin \mathit{fv}(\Gamma', A)$.
- If $\Gamma, x : \tau, \Gamma' \models \phi$ and τ is not a refinement type, then $x \notin \mathit{fv}(\Gamma', \phi)$ and $\Gamma, \Gamma' \models \phi$.
- If $\alpha \notin \mathit{fv}(\Gamma', \phi)$ and $\Gamma, \alpha, \Gamma' \vdash \phi$, then $\Gamma, \Gamma' \vdash \phi$.
- If $\alpha \notin \mathit{fv}(\Gamma', \phi)$ and $\Gamma, \alpha, \Gamma' \models \phi$, then $\Gamma, \Gamma' \models \phi$.

Assumption 34.

- $\vdash \mathit{ty}_{cps}(p)$ for all p.

Lemma 35 (Weakening)**.** *Assume that* $\vdash \Gamma_1, \Gamma_2, \Gamma_3$.

- *If* $\Gamma_1, \Gamma_3 \vdash \tau$*, then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \tau$.
- *If* $\Gamma_1, \Gamma_3 \vdash c : \tau$*, then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash c : \tau$.
- *If* $\Gamma_1, \Gamma_3 \vdash \tau_1 <: \tau_2$*, then* $\Gamma_1, \Gamma_2, \Gamma_3 \vdash \tau_1 <: \tau_2$.

Proof. By induction on the derivation. Assumption 33 is used. □

Lemma 36 (Narrowing)**.** *Assume that* $\Gamma \vdash \tau_1 <: \tau_2$.

- *If* $\vdash \Gamma, x : \tau_1, \Gamma'$ *and* $\Gamma, x : \tau_2, \Gamma' \vdash \tau$*, then* $\Gamma, x : \tau_1, \Gamma' \vdash \tau$.
- *If* $\vdash \Gamma, x : \tau_1, \Gamma'$ *and* $\Gamma, x : \tau_2, \Gamma' \vdash c : \tau$*, then* $\Gamma, x : \tau_1, \Gamma' \vdash c : \tau$.
- *If* $\Gamma, x : \tau_2, \Gamma' \vdash \tau_1 <: \tau_2$*, then* $\Gamma, x : \tau_1, \Gamma' \vdash \tau_1 <: \tau_2$.

Proof. By induction on the derivation. Assumption 33 is used. □

Lemma 37 (Remove unused type bindings)**.**

- *If* $x \notin \mathit{fv}(\Gamma')$ *and* $\vdash \Gamma, x : \tau_0, \Gamma'$*, then* $\vdash \Gamma, \Gamma'$.
- *If* $x \notin \mathit{fv}(\Gamma', \tau)$ *and* $\Gamma, x : \tau_0, \Gamma' \vdash \tau$*, then* $\Gamma, \Gamma' \vdash \tau$.

Proof. By induction on the derivation. The case for (WTc-Rfn) uses Assumption 33. □

Lemma 38 (Variables of non-refinement types do not apper in types)**.** *Assume that* τ_0 *is not a refinement type.*

- *If* $\vdash \Gamma, x : \tau_0, \Gamma'$*, then* $x \notin \mathit{fv}(\Gamma')$.
- *If* $\Gamma, x : \tau_0, \Gamma' \vdash \tau$*, then* $x \notin \mathit{fv}(\Gamma', \tau)$.

Proof. By induction on the derivation. The case for (WTc-Rfn) uses Assumption 33. □

Lemma 39 (Remove non-refinement type bindings)**.** *Assume that* τ_0 *is not a refinement type.*

1. *If* $\vdash \Gamma, x : \tau_0, \Gamma'$*, then* $\vdash \Gamma, \Gamma'$.
2. *If* $\Gamma, x : \tau_0, \Gamma' \vdash \tau$*, then* $\Gamma, \Gamma' \vdash \tau$.
3. *If* $x \notin \mathit{fv}(c)$ *and* $\Gamma, x : \tau_0, \Gamma' \vdash c : \tau$*, then* $\Gamma, \Gamma' \vdash c : \tau$.
4. *If* $\Gamma, x : \tau_0, \Gamma' \vdash \tau_1 <: \tau_2$*, then* $\Gamma, \Gamma' \vdash \tau_1 <: \tau_2$.

Proof.

1. Immediate by Lemma 38 and 37.
2. Immediate by Lemma 38 and 37.

3. By induction on the derivation. The case for (Tc-PApp) uses Assumption 33.

4. By induction on the derivation. The case for (Sc-Rfn) uses Assumption 33.

□

Lemma 40 (Remove unused type variable bindings)**.**

- *If* $\alpha \notin fv(\Gamma')$ *and* $\vdash \Gamma, \alpha, \Gamma'$*, then* $\vdash \Gamma, \Gamma'$*.*
- *If* $\alpha \notin fv(\Gamma', \tau)$ *and* $\Gamma, \alpha, \Gamma' \vdash \tau$*, then* $\Gamma, \Gamma' \vdash \tau$*.*
- *If* $\alpha \notin fv(\Gamma', \tau_1, \tau_2)$ *and* $\Gamma, \alpha, \Gamma' \vdash \tau_1 <: \tau_2$*, then* $\Gamma, \Gamma' \vdash \tau_1 <: \tau_2$*.*

Proof. By induction on the derivation. The case for (WTSc-Rfn) and (Sc-Rfn) uses Assumption 33. □

Lemma 41 (Substitution)**.** *Assume that* $\Gamma \vdash v : \tau_0$*.*

- *If* $\vdash \Gamma, x : \tau_0, \Gamma'$*, then* $\vdash \Gamma, \Gamma'[v/x]$*.*
- *If* $\Gamma, x : \tau_0, \Gamma' \vdash \tau$*, then* $\Gamma, \Gamma'[v/x] \vdash \tau[v/x]$*.*
- *If* $\Gamma, x : \tau_0, \Gamma' \vdash c : \tau$*, then* $\Gamma, \Gamma'[v/x] \vdash c[v/x] : \tau[v/x]$*.*
- *If* $\Gamma, x : \tau_0, \Gamma' \vdash \tau_1 <: \tau_2$*, then* $\Gamma, \Gamma'[v/x] \vdash \tau_1[v/x] <: \tau_2[v/x]$*.*

Proof. By induction on the derivation. Assumption 33 is used. □

Lemma 42 (Predicate substitution)**.** *Assume that* $\Gamma \vdash A : \widetilde{B}$*.*

- *If* $\vdash \Gamma, X : \widetilde{B}, \Gamma'$*, then* $\vdash \Gamma, \Gamma'[A/X]$*.*
- *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash \tau$*, then* $\Gamma, \Gamma'[A/X] \vdash \tau[A/X]$*.*
- *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash c : \tau$*, then* $\Gamma, \Gamma'[A/X] \vdash c[A/X] : \tau[A/X]$*.*
- *If* $\Gamma, X : \widetilde{B}, \Gamma' \vdash \tau_1 <: \tau_2$*, then* $\Gamma, \Gamma'[A/X] \vdash \tau_1[A/X] <: \tau_2[A/X]$*.*

Proof. By induction on the derivation. Assumption 33 is used. □

Lemma 43 (Type substitution)**.** *Assume that* $\Gamma \vdash \tau_0$*.*

- *If* $\vdash \Gamma, \alpha, \Gamma'$*, then* $\vdash \Gamma, \Gamma'[\tau_0/\alpha]$*.*
- *If* $\Gamma, \alpha, \Gamma' \vdash \tau$*, then* $\Gamma, \Gamma'[\tau_0/\alpha] \vdash \tau[\tau_0/\alpha]$*.*
- *If* $\Gamma, \alpha, \Gamma' \vdash c : \tau$*, then* $\Gamma, \Gamma'[\tau_0/\alpha] \vdash c[\tau_0/\alpha] : \tau[\tau_0/\alpha]$*.*
- *If* $\Gamma, \alpha, \Gamma' \vdash \tau_1 <: \tau_2$*, then* $\Gamma, \Gamma'[\tau_0/\alpha] \vdash \tau_1[\tau_0/\alpha] <: \tau_2[\tau_0/\alpha]$*.*

Proof. By induction on the derivation. Assumption 33 is used. □

Lemma 44 (Well-formedness of typing contexts from that of types)**.** *If* $\Gamma \vdash \tau$*, then* $\vdash \Gamma$*.*

Proof. By induction on the derivation. The case for (WTc-Rfn) uses Assumption 33. □

Lemma 45 (Well-formedness of types from typings)**.** *If* $\Gamma \vdash c : \tau$*, then* $\Gamma \vdash \tau$*.*

Proof. By induction on the derivation.

Case (Tc-CVar)**:** By Assumption 33.

Case (Tc-Var)**:** By Lemma 35.

Case (Tc-Prim)**:** By Assumption 34 and Lemma 35.

Case (Tc-Fun)**:** By the IH, Lemma 39, and (WTc-Fun).

Case (Tc-App)**:** By the IH, inversion, and Lemma 41.

Case (Tc-TAbs)**:** By the IH and (WTc-TPoly).

Case (Tc-TApp)**:** By the IH, inversion, and Lemma 43.

Case (TC-PABS)**:** By the IH and (WTC-PPOLY).

Case (TC-PAPP)**:** By the IH, inversion, and Lemma 42.

Case (TC-IF)**:** By the IH and Lemma 37.

Case (TC-ASCR) **and** (TC-SUB)**:** Immediate.

□

Lemma 46 (Reflexivity)**.** *If* $\Gamma \vdash \tau$*, then* $\Gamma \vdash \tau <: \tau$.

Proof. By induction on the derivation. The case for (WTC-RFN) uses Assumption 33. □

Lemma 47 (Transitivity)**.** *If* $\Gamma \vdash \tau_1 <: \tau_2$ *and* $\Gamma \vdash \tau_2 <: \tau_3$*, then* $\Gamma \vdash \tau_1 <: \tau_3$.

Proof. By induction on the structure of τ_2. Assumption 33, Lemma 36, and 35 are used. □

Lemma 48 (Inversion)**.**

- *If* $\Gamma \vdash x : \tau$*, then either*
  - $\vdash \Gamma$ *and* $\Gamma \vdash \{z : B \mid z = x\} <: \tau$ *(if* $\Gamma(x) = \{z : B \mid \phi\}$ *for some* z, B *and* ϕ*)*
  - $\vdash \Gamma$ *and* $\Gamma \vdash \Gamma(x) <: \tau$ *(otherwise)*
- *If* $\Gamma \vdash p : \tau$*, then* $\vdash \Gamma$ *and* $\Gamma \vdash ty_{cps}(p) <: \tau$.
- *If* $\Gamma \vdash \mathbf{rec}(f : (x : \tau_1) \to \tau_2, x : \tau_1).c : \tau$*, then* $\Gamma, f : (x : \tau_1) \to \tau_2, x : \tau_1 \vdash c : \tau_2$ *and* $\Gamma \vdash (x : \tau_1) \to \tau_2 <: \tau$.
- *If* $\Gamma \vdash \Lambda\alpha.c : \tau$*, then* $\Gamma, \alpha \vdash c : \tau'$ *and* $\Gamma \vdash \forall\alpha.\tau' <: \tau$ *for some* τ'.
- *If* $\Gamma \vdash \{(\mathsf{op}_i = v_i)_i\} : \tau$*, then* $(\Gamma \vdash v_i : \tau_i)_i$ *and* $\Gamma \vdash \{\mathsf{op}_i : \tau_i\} <: \tau$ *for some* $(\tau_i)_i$.
- *If* $\Gamma \vdash c\ v : \tau$*, then* $\Gamma \vdash c : (x : \tau_1) \to \tau_2$*,* $\Gamma \vdash v : \tau_1$ *and* $\Gamma \vdash \tau_2[v/x] <: \tau$ *for some* x, τ_1 *and* τ_2.
- *If* $\Gamma \vdash c\ \widetilde{A} : \tau$*, then* $\Gamma \vdash c : \forall\widetilde{X : \widetilde{B}}.\tau'$*,* $\Gamma \vdash \widetilde{A : \widetilde{B}}$ *and* $\Gamma \vdash \tau'[\widetilde{A/X}] <: \tau$ *for some* $\widetilde{X}, \widetilde{\widetilde{B}}$ *and* τ'.
- *If* $\Gamma \vdash c\ \tau' : \tau$*, then* $\Gamma \vdash c : \forall\alpha.\tau_1$*,* $\Gamma \vdash \tau'$ *and* $\Gamma \vdash \tau_1[\tau'/\alpha] <: \tau$ *for some* α *and* τ_1.
- *If* $\Gamma \vdash v\#\mathsf{op} : \tau$*, then* $\Gamma \vdash v : \{\ldots, \mathsf{op} : \tau, \ldots\}$.
- *If* $\Gamma \vdash (c : \tau') : \tau$*, then* $\Gamma \vdash c : \tau'$ *and* $\Gamma \vdash \tau' <: \tau$.
- *If* $\Gamma \vdash \mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : \tau$*, then* $\Gamma \vdash v : \{z : \mathrm{bool} \mid \phi\}$*,* $\Gamma, v = \mathbf{true} \vdash c_1 : \tau'$*,* $\Gamma, v = \mathbf{false} \vdash c_2 : \tau'$*, and* $\Gamma \vdash \tau' <: \tau$ *for some* z, ϕ *and* τ'.

Proof. By induction on the derivation. Lemma 46 and 47 are used. □

Lemma 49 (Inversion for CPS-transformed computations)**.** *If* $\Gamma \vdash \Lambda\alpha.\lambda h : \tau_h.\lambda k : \tau_k.c : \tau$ *and neither* τ_h *nor* τ_k *is a refinement type, then there exists some* τ' *such that*

- $\Gamma, \alpha, h : \tau_h, k : \tau_k \vdash c : \tau'$ *and*
- $\Gamma \vdash \forall\alpha.\tau_h \to \tau_k \to \tau' <: \tau$ *.*

Proof. By Lemma 48, (SC-POLY), (SC-FUN), and Lemma 47. □

Lemma 50 (Inversion for the specific form of application)**.** *If* $\Gamma \vdash c\ \tau_0\ v_1\ v_2 : \tau$*, then there exist some* τ', τ_1*, and* τ_2 *such that*

- $\Gamma \vdash c : \tau'$*,*
- $\Gamma \vdash v_1 : \tau_1$*, and*
- $\Gamma \vdash v_2 : \tau_2$ *.*

In addition, if $\Gamma \vdash \tau_1' <: \tau_1$ *and* $\Gamma \vdash \tau_2' <: \tau_2$ *for some* τ_1' *and* τ_2' *and neither* τ_1' *nor* τ_2' *is a refinement type, then* $\Gamma \vdash \tau' <: \forall\alpha.\tau_1' \to \tau_2' \to \tau$ *where* α *is fresh.*

Proof. The first half is by Lemma 48. The second half is by Lemma 45, 35 and 47 with the results of the first half. □

G.2 Forward type preservation

Assumption 51.

- If $\Gamma \vdash \phi$, then $[\![\Gamma]\!] \vdash \phi$.
- If $\Gamma \vdash A : \widetilde{B}$, then $[\![\Gamma]\!] \vdash A : \widetilde{B}$.
- If $\Gamma \models \phi$, then $[\![\Gamma]\!] \models \phi$.

Assumption 52.

- $[\![ty(p)]\!] = ty_{cps}([\![p]\!])$.
- If $ty(p) = \{x : B \mid \phi\}$ for some x, B and ϕ, then $[\![p]\!] = p$.

Lemma 53 (CPS transformation preserves free variables in types)**.**

- $fv([\![T]\!]) = fv(T)$.
- $fv([\![C]\!]) = fv(C)$.
- $fv([\![\Sigma]\!]) = fv(\Sigma)$.

Proof. By simultaneous induction on the structure of types. □

Lemma 54 (CPS transformation is homomorphic for substitution)**.**

- $[\![T[v/x]]\!] = [\![T]\!][[\![v]\!]/x]$.
- $[\![C[v/x]]\!] = [\![C]\!][[\![v]\!]/x]$.
- $[\![\Sigma[v/x]]\!] = [\![\Sigma]\!][[\![v]\!]/x]$.
- $[\![T[A/X]]\!] = [\![T]\!][A/X]$.
- $[\![C[A/X]]\!] = [\![C]\!][A/X]$.
- $[\![\Sigma[A/X]]\!] = [\![\Sigma]\!][A/X]$.

Proof. By simultaneous induction on the structure of types. The case for $T = \{x : B \mid \phi\}$ uses Assumption 52. □

Lemma 55 (CPS transformation preserves well-formedness)**.**

- *If* $\vdash \Gamma$*, then* $\vdash [\![\Gamma]\!]$.
- *If* $\Gamma \vdash T$*, then* $[\![\Gamma]\!] \vdash [\![T]\!]$.
- *If* $\Gamma \vdash C$*, then* $[\![\Gamma]\!] \vdash [\![C]\!]$.
- *If* $\Gamma \vdash \Sigma$*, then* $[\![\Gamma]\!] \vdash [\![\Sigma]\!]$.

Proof. By simultaneous induction on the derivations. Lemma 35 is used. The case for (WT-Rfn) uses Assumption 51. □

Lemma 56 (CPS transformation preserves subtyping)**.**

- *If* $\Gamma \vdash T_1 <: T_2$*, then* $[\![\Gamma]\!] \vdash [\![T_1]\!] <: [\![T_2]\!]$.
- *If* $\Gamma \vdash C_1 <: C_2$*, then* $[\![\Gamma]\!] \vdash [\![C_1]\!] <: [\![C_2]\!]$.
- *If* $\Gamma \vdash \Sigma_1 <: \Sigma_2$*, then* $[\![\Gamma]\!] \vdash [\![\Sigma_1]\!] <: [\![\Sigma_2]\!]$.

Proof. By simultaneous induction on the derivations. Lemma 35 is used. The case for (S-Rfn) uses Assumption 51. □

Theorem 57 (Forward type preservation)**.**

1. *If* $\Gamma \vdash v : T$*, then* $[\![\Gamma]\!] \vdash [\![v]\!] : [\![T]\!]$.
2. *If* $\Gamma \vdash c : C$*, then* $[\![\Gamma]\!] \vdash [\![c]\!] : [\![C]\!]$.

Proof. By simultaneous induction on the typing derivation of the source language.

1. **Case** (T-CVar)**:** By Lemma 55, definition of CPS transformation of typing contexts, and (Tc-CVar).
   Case (T-Var)**:** By Lemma 55, definition of CPS transformation of typing contexts, and (Tc-Var).
   Case (T-Prim)**:** By Lemma 55, Assumption 52, and (Tc-Prim).
   Case (T-Fun)**:** By the IH and (Tc-Fun).
   Case (T-VSub)**:** By the IH, Lemma 56, Lemma 55 and (Tc-Sub).

2. **Case** (T-Ret)**:** we have
   - $c = \textbf{return}\ v$,
   - $C = \{\} \triangleright T \ / \ \square$, and
   - $\Gamma \vdash v : T$

   for some v and T. Then, we have
   - $[\![c]\!] = \Lambda\alpha.\lambda h : \{\}.\lambda k : [\![T]\!] \to \alpha.k\ [\![v]\!]$ and
   - $[\![C]\!] = \forall\alpha.\{\} \to ([\![T]\!] \to \alpha) \to \alpha$.

   By the IH, we have
   - $[\![\Gamma]\!] \vdash [\![v]\!] : [\![T]\!]$.

   We have the conclusion by the following derivation with Lemma 35:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_{\alpha,h,k} \vdash k : [\![T]\!] \to \alpha}\,(\textsc{Tc-Var}) \qquad \Gamma_{\alpha,h,k} \vdash [\![v]\!] : [\![T]\!]}{[\![\Gamma]\!], \alpha, h : \{\}, k : [\![T]\!] \to \alpha \vdash k\ [\![v]\!] : \alpha}\,(\textsc{Tc-App})}{[\![\Gamma]\!], \alpha, h : \{\} \vdash \lambda k [\![T]\!] \to \alpha.k\ [\![v]\!] : ([\![T]\!] \to \alpha) \to \alpha}\,(\textsc{Tc-Lam})}{[\![\Gamma]\!], \alpha \vdash \lambda h : \{\}.\lambda k : [\![T]\!] \to \alpha.k\ [\![v]\!] : \{\} \to ([\![T]\!] \to \alpha) \to \alpha}\,(\textsc{Tc-Lam})}{[\![\Gamma]\!] \vdash \Lambda\alpha.\lambda h : \{\}.\lambda k : [\![T]\!] \to \alpha.k\ [\![v]\!] : \forall\alpha.\{\} \to ([\![T]\!] \to \alpha) \to \alpha}\,(\textsc{Tc-TAbs})$$

   where $\Gamma_{\alpha,h,k} \stackrel{\text{def}}{=} [\![\Gamma]\!], \alpha, h : \{\}, k : [\![T]\!] \to \alpha$.

   Case (T-App)**:** By the IH, Lemma 54 and (Tc-App).
   Case (T-If)**:** By the IH, (Tc-If) and (Tc-Ascr).
   Case (T-CSub)**:** similar to the case for (T-VSub).
   Case (T-LetP)**:** We have
   - $c = \textbf{let}\ x = c_1\ \textbf{in}\ c_2$,
   - $C = \Sigma \triangleright T_2 \ / \ \square$,
   - $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ / \ \square$,
   - $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \ / \ \square$, and
   - $x \notin \mathit{fv}(T_2) \cup \mathit{fv}(\Sigma)$

   for some x, c_1, c_2, Σ, T_1 and T_2. Then we have
   - $[\![c]\!] = \Lambda\alpha.\lambda h : [\![\Sigma]\!].\lambda k : [\![T_2]\!] \to \alpha.[\![c_1]\!]\ \alpha\ h\ (\lambda x : [\![T_1]\!].[\![c_2]\!]\ \alpha\ h\ k)$ and
   - $[\![C]\!] = \forall\alpha.[\![\Sigma]\!] \to ([\![T_2]\!] \to \alpha) \to \alpha$.

   By Lemma 53, we have
   - $x \notin \mathit{fv}([\![T_2]\!]) \cup \mathit{fv}([\![\Sigma]\!])$.

   Also, by the IHs, we have
   - $[\![\Gamma]\!] \vdash [\![c_1]\!] : \forall\beta.[\![\Sigma]\!] \to ([\![T_1]\!] \to \beta) \to \beta$ and
   - $[\![\Gamma]\!], x : [\![T_1]\!] \vdash [\![c_2]\!] : \forall\gamma.[\![\Sigma]\!] \to ([\![T_2]\!] \to \gamma) \to \gamma$.

   We have the conclusion by the following derivations with Lemma 35:

$$(A): \quad \dfrac{\dfrac{\dfrac{\Gamma_{\alpha,h,k} \vdash [\![c_1]\!] : \forall\beta.[\![\Sigma]\!] \to ([\![T_1]\!] \to \beta) \to \beta \qquad \Gamma_{\alpha,h,k} \vdash \alpha}{\Gamma_{\alpha,h,k} \vdash [\![c_1]\!]\ \alpha : [\![\Sigma]\!] \to ([\![T_1]\!] \to \alpha) \to \alpha}\,(\textsc{Tc-TApp}) \qquad \dfrac{}{\Gamma_{\alpha,h,k} \vdash h : [\![\Sigma]\!]}\,(\textsc{Tc-Var})}{\Gamma_{\alpha,h,k} \vdash [\![c_1]\!]\ \alpha\ h : ([\![T_1]\!] \to \alpha) \to \alpha}\,(\textsc{Tc-App})}{[\![\Gamma]\!]_{\alpha,h,k} \vdash [\![c_1]\!]\ \alpha\ h : ([\![T_1]\!] \to \alpha) \to \alpha}\,(\textsc{Tc-App})$$

$$(B): \quad \dfrac{\dfrac{\Gamma_{\alpha,h,k,x} \vdash [\![c_2]\!] : \forall\gamma.[\![\Sigma]\!] \to ([\![T_2]\!] \to \gamma) \to \gamma \qquad \Gamma_{\alpha,h,k,x} \vdash \alpha}{\Gamma_{\alpha,h,k,x} \vdash [\![c_2]\!]\ \alpha : [\![\Sigma]\!] \to ([\![T_2]\!] \to \alpha) \to \alpha}\,(\textsc{Tc-TApp}) \qquad \dfrac{}{\Gamma_{\alpha,h,k,x} \vdash h : [\![\Sigma]\!]}\,(\textsc{Tc-Var})}{\Gamma_{\alpha,h,k,x} \vdash [\![c_2]\!]\ \alpha\ h : ([\![T_2]\!] \to \alpha) \to \alpha}\,(\textsc{Tc-App})$$

$$
\dfrac{\dfrac{(A) \quad \dfrac{\dfrac{(B) \quad \dfrac{}{\Gamma_{\alpha,h,k,x} \vdash k : \llbracket T_2 \rrbracket \rightarrow \alpha}\,(\textsc{Tc-Var})}{\Gamma_{\alpha,h,k}, x : \llbracket T_1 \rrbracket \vdash \llbracket c_2 \rrbracket\ \alpha\ h\ k : \alpha}\,(\textsc{Tc-App})}{\Gamma_{\alpha,h,k} \vdash \lambda x : \llbracket T_1 \rrbracket.\llbracket c_2 \rrbracket\ \alpha\ h\ k : \llbracket T_1 \rrbracket \rightarrow \alpha}\,(\textsc{Tc-Fun})}{\dfrac{\llbracket \Gamma \rrbracket, \alpha, h : \llbracket \Sigma \rrbracket, k : \llbracket T_2 \rrbracket \rightarrow \alpha \vdash \llbracket c_1 \rrbracket\ \alpha\ h\ (\lambda x : \llbracket T_1 \rrbracket.\llbracket c_2 \rrbracket\ \alpha\ h\ k) : \alpha}{\dfrac{\llbracket \Gamma \rrbracket, \alpha, h : \llbracket \Sigma \rrbracket \vdash \lambda k : \llbracket T_2 \rrbracket \rightarrow \alpha.\llbracket c_1 \rrbracket\ \alpha\ h\ (\lambda x : \llbracket T_1 \rrbracket.\llbracket c_2 \rrbracket\ \alpha\ h\ k) : (\llbracket T_2 \rrbracket \rightarrow \alpha) \rightarrow \alpha}{\llbracket \Gamma \rrbracket, \alpha \vdash \lambda h : \llbracket \Sigma \rrbracket.\lambda k : \llbracket T_2 \rrbracket \rightarrow \alpha.\llbracket c_1 \rrbracket\ \alpha\ h\ (\lambda x : \llbracket T_1 \rrbracket.\llbracket c_2 \rrbracket\ \alpha\ h\ k) : \llbracket \Sigma \rrbracket \rightarrow (\llbracket T_2 \rrbracket \rightarrow \alpha) \rightarrow \alpha}\,(\textsc{Tc-Fun})}\,(\textsc{Tc-Fun})}\,(\textsc{Tc-App})}{\llbracket \Gamma \rrbracket \vdash \Lambda \alpha.\lambda h : \llbracket \Sigma \rrbracket.\lambda k : \llbracket T_2 \rrbracket \rightarrow \alpha.\llbracket c_1 \rrbracket\ \alpha\ h\ (\lambda x : \llbracket T_1 \rrbracket.\llbracket c_2 \rrbracket\ \alpha\ h\ k) : \forall \alpha.\llbracket \Sigma \rrbracket \rightarrow (\llbracket T_2 \rrbracket \rightarrow \alpha) \rightarrow \alpha}\,(\textsc{Tc-TAbs})
$$

where $\Gamma_{\alpha,h,k} \stackrel{\text{def}}{=} \llbracket \Gamma \rrbracket, \alpha, h : \llbracket \Sigma \rrbracket, k : \llbracket T_2 \rrbracket \rightarrow \alpha$ and $\Gamma_{\alpha,h,k,x} \stackrel{\text{def}}{=} \Gamma_{\alpha,h,k}, x : \llbracket T_1 \rrbracket$.

Case (T-LetIp)**:** We have

- $c = \textbf{let } x = c_1 \textbf{ in } c_2$,
- $C = \Sigma \triangleright T_2 \ / \ (\forall z.C_0) \Rightarrow C_2$,
- $\Gamma \vdash c_1 : \Sigma \triangleright T_1 \ / \ (\forall x.C_1) \Rightarrow C_2$,
- $\Gamma, x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \ / \ (\forall z.C_0) \Rightarrow C_1$, and
- $x \notin fv(T_2) \cup fv(\Sigma) \cup (fv(C_0) \setminus \{z\})$

for some $x, z, c_1, c_2, \Sigma, T_1, T_2, C_0, C_1$ and C_2. Then we have

- $\llbracket c \rrbracket = \Lambda \alpha.\lambda h : \llbracket \Sigma \rrbracket.\lambda k : (z : \llbracket T_2 \rrbracket) \rightarrow \llbracket C_0 \rrbracket.\llbracket c_1 \rrbracket\ \llbracket C_2 \rrbracket\ h\ (\lambda x : \llbracket T_1 \rrbracket.\llbracket c_2 \rrbracket\ \llbracket C_1 \rrbracket\ h\ k)$ and
- $\llbracket C \rrbracket = \forall \alpha.\llbracket \Sigma \rrbracket \rightarrow ((z : \llbracket T_2 \rrbracket) \rightarrow \llbracket C_0 \rrbracket) \rightarrow \llbracket C_2 \rrbracket$.

By Lemma 53, we have

- $x \notin fv(\llbracket T_2 \rrbracket) \cup fv(\llbracket \Sigma \rrbracket) \cup (fv(\llbracket C_0 \rrbracket) \setminus \{z\})$.

Also, by the IHs, we have

- $\llbracket \Gamma \rrbracket \vdash \llbracket c_1 \rrbracket : \forall \alpha.\llbracket \Sigma \rrbracket \rightarrow ((x : \llbracket T_1 \rrbracket) \rightarrow \llbracket C_1 \rrbracket) \rightarrow \llbracket C_2 \rrbracket$ and
- $\llbracket \Gamma \rrbracket, x : \llbracket T_1 \rrbracket \vdash \llbracket c_2 \rrbracket : \forall \alpha.\llbracket \Sigma \rrbracket \rightarrow ((z : \llbracket T_2 \rrbracket) \rightarrow \llbracket C_0 \rrbracket) \rightarrow \llbracket C_1 \rrbracket$.

We have the conclusion by a straightforward derivation like the case for (T-LetP) using those judgements shown so far and Lemma 35.

Case (T-Op)**:** (In this case, we use Lemma 54 frequently and implicitly.)
We have

- $c = \mathsf{op}\ v$,
- $C = \Sigma \triangleright T_2[\widetilde{A/X}][v/x] \ / \ (\forall y.C_1[\widetilde{A/X}][v/x]) \Rightarrow C_2[\widetilde{A/X}][v/x]$,
- $\Sigma \ni \mathsf{op} : \forall \widetilde{X : B}.(x : T_1) \rightarrow ((y : T_2) \rightarrow C_1) \rightarrow C_2$,
- $\Gamma \vdash \Sigma$,
- $\Gamma \vdash \widetilde{A : B}$, and
- $\Gamma \vdash v : T_1[\widetilde{A/X}]$

for some $x, y, v, \widetilde{X}, \widetilde{A}, \widetilde{\widetilde{B}}, \Sigma, T_1, T_2, C_1$ and C_2. Then, we have

- $\llbracket c \rrbracket = \Lambda \alpha.\lambda h : \llbracket \Sigma \rrbracket.\lambda k : (y : \llbracket T_2 \rrbracket[\widetilde{A/X}][\llbracket v \rrbracket/x] \rightarrow \llbracket C_1 \rrbracket[\widetilde{A/X}][\llbracket v \rrbracket/x]).$
$h\#\mathsf{op}\ \widetilde{A}\ \llbracket v \rrbracket\ (\lambda y' : \llbracket T_2 \rrbracket[\widetilde{A/X}][\llbracket v \rrbracket/x].k\ y')$,
- $\llbracket C \rrbracket = \forall \alpha.\llbracket \Sigma \rrbracket \rightarrow ((y : \llbracket T_2 \rrbracket[\widetilde{A/X}][\llbracket v \rrbracket/x]) \rightarrow \llbracket C_1 \rrbracket[\widetilde{A/X}][\llbracket v \rrbracket/x]) \rightarrow \llbracket C_2 \rrbracket[\widetilde{A/X}][\llbracket v \rrbracket/x]$, and
- $\llbracket \Sigma \rrbracket \ni \mathsf{op} : \forall \widetilde{X : B}.(x : \llbracket T_1 \rrbracket) \rightarrow ((y : \llbracket T_2 \rrbracket) \rightarrow \llbracket C_1 \rrbracket) \rightarrow \llbracket C_2 \rrbracket$.

Also, by the IHs, we have

- $\llbracket \Gamma \rrbracket \vdash \llbracket v \rrbracket : \llbracket T_1 \rrbracket[\widetilde{A/X}]$.

By Assumption 51, we have

- $\widetilde{\llbracket \Gamma \rrbracket \vdash A : B}$.

We have the conclusion by a straightforward derivation like the cases for (T-Ret) and (T-LetP) using those judgements shown so far and Lemma 35.

Case (T-Hndl)**:** We have

- $c = \textbf{with } h \textbf{ handle } c_0$,
- $h = \{\textbf{return } x_r \mapsto c_r, (\mathsf{op}_i(x_i, k_i) \mapsto c_i)_i\}$,
- $\Gamma \vdash c_0 : \Sigma_0 \triangleright T_r \ / \ (\forall x_r.C_1) \Rightarrow C$,

- $\Gamma, x_r : T_r \vdash c_r : C_1$,
- $\left(\Gamma, \widetilde{X_i : \widetilde{B}_i}, x_i : T_{i1}, k_i : (y_i : T_{i2}) \to C_{i1} \vdash c_i : C_{i2}\right)_i$, and
- $\Sigma_0 = \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : T_{i1}) \to ((y_i : T_{i2}) \to C_{i1}) \to C_{2i})_i\}$

Then, we have

- $[\![c]\!] = [\![c_0]\!]\ [\![C]\!]\ [\![h^{ops}]\!]\ [\![h^{ret}]\!]$,
- $[\![h^{ret}]\!] = \lambda x_r : [\![T_r]\!].[\![c_r]\!]$,
- $[\![h^{ops}]\!] = \{(\mathsf{op}_i = \Lambda \widetilde{X_i : \widetilde{B}_i}.\lambda x_i : [\![T_{i1}]\!].\lambda k_i : (y_i : [\![T_{i2}]\!]) \to [\![C_{i1}]\!].[\![c_i]\!])_i\}$, and
- $[\![\Sigma_0]\!] = \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : [\![T_{i1}]\!]) \to ((y_i : [\![T_{i2}]\!]) \to [\![C_{i1}]\!]) \to [\![C_{i2}]\!])_i\}$.

Also, by the IHs, we have

- $[\![\Gamma]\!] \vdash [\![c_0]\!] : \forall \alpha.[\![\Sigma_0]\!] \to ((x_r : [\![T_r]\!]) \to [\![C_1]\!]) \to [\![C]\!]$,
- $[\![\Gamma]\!], x_r : [\![T_r]\!] \vdash [\![c_r]\!] : [\![C_1]\!]$, and
- $\left([\![\Gamma]\!], \widetilde{X_i : \widetilde{B}_i}, x_i : [\![T_{i1}]\!], k_i : (y_i : [\![T_{i2}]\!]) \to [\![C_{i1}]\!] \vdash [\![c_i]\!] : [\![C_{i2}]\!]\right)_i$.

We have the conclusion by a straightforward derivation like the cases for (T-Ret) and (T-LetP) using those judgements shown so far and Lemma 35.

□

G.3 Backward type preservation

For the backward direction, we define some notations.

Definition 58. Γ is *cps-wellformed* if for all $(x : \tau) \in \Gamma$, it holds that $\tau = [\![T]\!]$ for some T.

Definition 59. *rmtv* is a function which removes all bindings of type variables from a typing context. Formally, it is defined as follows:

$$\mathit{rmtv}(\emptyset) \stackrel{\text{def}}{=} \emptyset \qquad \mathit{rmtv}(\Gamma, x : \tau) \stackrel{\text{def}}{=} \mathit{rmtv}(\Gamma), x : \tau$$
$$\mathit{rmtv}(\Gamma, X : \widetilde{B}) \stackrel{\text{def}}{=} \mathit{rmtv}(\Gamma), X : \widetilde{B} \qquad \mathit{rmtv}(\Gamma, \alpha) \stackrel{\text{def}}{=} \mathit{rmtv}(\Gamma)$$

Lemma 60 (CPS-wellformed target typing contexts have corresponding source ones)**.** *If Γ is cps-wellformed, then there exists some Γ' such that $[\![\Gamma']\!] = \mathit{rmtv}(\Gamma)$.*

Proof. By induction on the structure of Γ. □

Since the CPS transformation is injective, there is only one Γ' which satisfies the equation in Lemma 60. Therefore, we define a function $(\!|-|\!)$ that maps Γ to Γ':

Definition 61. Let Γ be a cps-wellformed typing context in the target language. We define $(\!|\Gamma|\!)$ to be the typing context in the source language such that $[\![(\!|\Gamma|\!)]\!] = \mathit{rmtv}(\Gamma)$.

Assumption 62. Assume that Γ is cps-wellformed.

- If $\Gamma \vdash \phi$, then $(\!|\Gamma|\!) \vdash \phi$.
- If $\Gamma \vdash A : \widetilde{B}$, then $(\!|\Gamma|\!) \vdash A : \widetilde{B}$.
- If $\Gamma \vDash \phi$, then $(\!|\Gamma|\!) \vDash \phi$.

Lemma 63 (Computation types in the specific form of subtyping are pure)**.** *If $\Gamma \vdash [\![C]\!] <: \forall \alpha.\tau_1 \to (\tau_2 \to \beta) \to \tau_4$ and $\beta \in \Gamma$, then $C = \Sigma \triangleright T \ / \ \square$ (for some Σ and T), and $\tau_4 = \beta$.*

Proof. Assume that $C = \Sigma \triangleright T \ / \ (\forall x.C_1) \Rightarrow C_2$ for some Σ, T, x, C_1 and C_2. Then, we have

$$\Gamma \vdash \forall \gamma.[\![\Sigma]\!] \to ((x : [\![T]\!]) \to [\![C_1]\!]) \to [\![C_2]\!] <: \forall \alpha.\tau_1 \to (\tau_2 \to \beta) \to \tau_4$$

where γ is fresh. By inversion, we have $\Gamma, \alpha, h : \tau_1, x : [\![T]\!] \vdash \beta <: [\![C_1]\!]$, that is, $\Gamma, \alpha, h : \tau_1, x : [\![T]\!] \vdash \beta <: \forall \delta.\tau_5$ for some τ_5 and δ. This is contradictory since there is no subtyping rule for such a judgment.

Therefore, $C = \Sigma \triangleright T \ / \ \square$ for some Σ and T. In this case, we have

$$\Gamma \vdash \forall \gamma.[\![\Sigma]\!] \to ([\![T]\!] \to \gamma) \to \gamma <: \forall \alpha.\tau_1 \to (\tau_2 \to \beta) \to \tau_4$$

where γ is fresh. By inversion, we have

- $\Gamma, \alpha \vdash \tau_6$,
- $\Gamma, \alpha, h : \tau_1, x : [\![T]\!][\tau_6/\gamma] \vdash \beta <: \gamma[\tau_6/\gamma]$, and
- $\Gamma, \alpha, h : \tau_1, x : [\![T]\!][\tau_6/\gamma] \vdash \gamma[\tau_6/\gamma] <: \tau_4$

for some τ_6. The second judgment can be derived by only (Sc-TVar) where $\gamma[\tau_6/\gamma] = \beta$. Therefore, the third judgment becomes $\Gamma, \alpha, h : \tau_1, x : [\![T]\!][\tau_6/\gamma] \vdash \beta <: \tau_4$, which can be derived by only (Sc-TVar) where $\tau_4 = \beta$. □

Lemma 64 (Computation types can be assumed to be impure)**.** *If $\Gamma \vdash c : C$, then w.l.o.g., we can assume that $C = \Sigma \triangleright T \;/\; (\forall x.C_1) \Rightarrow C_2$ for some Σ, T, x, C_1 and C_2.*

Proof.

Case $C = \Sigma \triangleright T \;/\; (\forall x.C_1) \Rightarrow C_2$: Immediate.

Case $C = \Sigma \triangleright T \;/\; \square$: It holds that $\Gamma \vdash \Sigma \triangleright T \;/\; \square <: \Sigma \triangleright T \;/\; (\forall x.C_0) \Rightarrow C_0$ for any C_0 such that $\Gamma \vdash C_0$. Therefore, by subsumption we have $\Gamma \vdash c : \Sigma \triangleright T \;/\; (\forall x.C_0) \Rightarrow C_0$.

□

Lemma 65 (Backward preservation on well-formedness)**.** *Assume that Γ is cps-wellformed.*

1. *If $\vdash \Gamma$, then $\vdash (\!|\Gamma|\!)$.*
2. *If $\Gamma \vdash [\![T]\!]$, then $(\!|\Gamma|\!) \vdash T$.*
3. *If $\Gamma \vdash [\![C]\!]$, then $(\!|\Gamma|\!) \vdash C$.*
4. *If $\Gamma \vdash [\![\Sigma]\!]$, then $(\!|\Gamma|\!) \vdash \Sigma$.*

Proof. By simultaneous induction on the derivation.

1. **Case (WEc-Empty):** Obvious since $(\!|\emptyset|\!) = \emptyset$.

   Case (WEc-Var): We have

   - $\Gamma = \Gamma', x : \tau$,
   - $\vdash \Gamma'$,
   - $x \notin dom(\Gamma')$, and
   - $\Gamma' \vdash \tau$

   for some Γ', x, and τ. By the IH, we have $\vdash (\!|\Gamma'|\!)$. Also, we have $x \notin dom((\!|\Gamma'|\!))$ since $dom(\Gamma') \supseteq dom((\!|\Gamma'|\!))$. Moreover, since Γ is cps-wellformed, $\tau = [\![T]\!]$ for some T. Then, by the IH, we have $(\!|\Gamma'|\!) \vdash T$. We have the conclusion by (WE-Var). (Note that $(\!|\Gamma|\!) = (\!|\Gamma', x : [\![T]\!]|\!) = (\!|\Gamma'|\!), x : T$.)

   Case (WEc-BVar): By the IH and (WE-BVar).

   Case (WEc-PVar): By the IH and (WE-PVar).

   Case (WEc-TVar): By the IH. Note that $(\!|\Gamma', \alpha|\!) = (\!|\Gamma'|\!)$.

2. Case analysis on T.

   Case $T = \{z : B \mid \phi\}$: By Assumption 62 and (WT-Rfn).

   Case $T = (x : T_1) \to C_1$: By the IHs and (WT-Fun).

3. Case analysis on C.

   Case $C = \Sigma \triangleright T \;/\; \square$: We have $[\![C]\!] = \forall \alpha. [\![\Sigma]\!] \to ([\![T]\!] \to \alpha) \to \alpha$ for some α. By inversion, we have

   - $\Gamma, \alpha \vdash [\![\Sigma]\!]$ and
   - $\Gamma, \alpha, h : [\![\Sigma]\!] \vdash [\![T]\!]$.

   By 9, we have

   - $\Gamma, \alpha \vdash [\![T]\!]$.

   By the IHs, we have

   - $(\!|\Gamma|\!) \vdash \Sigma$ and
   - $(\!|\Gamma|\!) \vdash T$.

   Also, by (WT-Pure), we have $(\!|\Gamma|\!) \mid T \vdash \square$. Then we have the conclusion by (WT-Comp).

Case $C = \Sigma \triangleright T \;/\; (\forall x.C_1) \Rightarrow C_2$**:** We have $[\![C]\!] = \forall\alpha.[\![\Sigma]\!] \to ((x : [\![T]\!]) \to [\![C_1]\!]) \to [\![C_2]\!]$. By inversion, we have

- $\Gamma, \alpha \vdash [\![\Sigma]\!]$,
- $\Gamma, \alpha, h : [\![\Sigma]\!] \vdash [\![T]\!]$,
- $\Gamma, \alpha, h : [\![\Sigma]\!], x : [\![T]\!] \vdash [\![C_1]\!]$, and
- $\Gamma, \alpha, h : [\![\Sigma]\!], k : (x : [\![T]\!]) \to [\![C_1]\!] \vdash [\![C_2]\!]$.

By 9, we have

- $\Gamma, \alpha \vdash [\![T]\!]$,
- $\Gamma, \alpha, x : [\![T]\!] \vdash [\![C_1]\!]$, and
- $\Gamma, \alpha \vdash [\![C_2]\!]$.

By the IHs, we have

- $(\!|\Gamma|\!) \vdash \Sigma$,
- $(\!|\Gamma|\!) \vdash T$,
- $(\!|\Gamma|\!), x : T \vdash C_1$, and
- $(\!|\Gamma|\!) \vdash C_2$.

Then we have the conclusion by (WT-ATM) and (WT-Comp).

4. By the IHs and (WT-Sig).

□

Lemma 66 (Backward preservation on subtyping)**.** *Assume that* Γ *is cps-wellformed.*

1. If $\Gamma \vdash [\![T_1]\!] <: [\![T_2]\!]$*, then* $(\!|\Gamma|\!) \vdash T_1 <: T_2$*.*

2. If $\Gamma \vdash [\![C_1]\!] <: [\![C_2]\!]$*, then* $(\!|\Gamma|\!) \vdash C_1 <: C_2$*.*

3. If $\Gamma \vdash [\![\Sigma_1]\!] <: [\![\Sigma_2]\!]$*, then* $(\!|\Gamma|\!) \vdash \Sigma_1 <: \Sigma_2$*.*

Proof. By simultaneous induction on the derivation.

1. Case analysis on T_1 and T_2.

Case $T_1 = \{z : B \mid \phi_1\}$ **and** $T_2 = \{z : B \mid \phi_2\}$**:** We have

- $[\![T_1]\!] = \{z : B \mid \phi_1\}$ and
- $[\![T_2]\!] = \{z : B \mid \phi_2\}$.

We have the conclusion by Assumption 62 and (S-Rfn).

Case $T_1 = (x : T_{10}) \to C_1$ **and** $T_1 = (x : T_{10}) \to C_1$**:** We have

- $[\![T_1]\!] = (x : [\![T_{10}]\!]) \to [\![C_1]\!]$ and
- $[\![T_2]\!] = (x : [\![T_{20}]\!]) \to [\![C_2]\!]$.

We have the conclusion by the IHs and (S-Fun).

Otherwise: Contradictory since there is no applicable rule.

2. Case analysis on C_1 and C_2.

Case $C_1 = \Sigma_1 \triangleright T_1 \;/\; \square$ **and** $C_2 = \Sigma_2 \triangleright T_2 \;/\; \square$**:** We have

- $[\![C_1]\!] = \forall\alpha.[\![\Sigma_1]\!] \to ([\![T_1]\!] \to \alpha) \to \alpha$ and
- $[\![C_2]\!] = \forall\beta.[\![\Sigma_2]\!] \to ([\![T_2]\!] \to \beta) \to \beta$

for some α and β. By inversion, we have

- $\Gamma, \beta \vdash \tau$,
- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!][\tau/\alpha]$ and
- $\Gamma, \beta, h : [\![\Sigma_2]\!] \vdash [\![T_1]\!] <: [\![T_2]\!][\tau/\alpha]$

for some τ. Since CPS-transformed types do not contain type variables, we have

- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!]$ and
- $\Gamma, \beta, h : [\![\Sigma_2]\!] \vdash [\![T_1]\!] <: [\![T_2]\!]$.

By 9, we have

- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!]$ and

- $\Gamma, \beta \vdash [\![T_1]\!] <: [\![T_2]\!]$.

By the IHs, we have

- $(\!|\Gamma|\!) \vdash \Sigma_2 <: \Sigma_1$ and
- $(\!|\Gamma|\!) \vdash T_1 <: T_2$.

Also, by (S-Pure), we have $(\!|\Gamma|\!) \mid T_1 \vdash \square <: \square$. Then we have the conclusion by (S-Comp).

Case $C_1 = \Sigma_1 \triangleright T_1 \ / \ (\forall x.C_{11}) \Rightarrow C_{12}$ **and** $C_2 = \Sigma_2 \triangleright T_2 \ / \ (\forall x.C_{21}) \Rightarrow C_{22}$**:** We have

- $[\![C_1]\!] = \forall \alpha.[\![\Sigma_1]\!] \to ((x : [\![T_1]\!]) \to [\![C_{11}]\!]) \to [\![C_{12}]\!]$ and
- $[\![C_2]\!] = \forall \beta.[\![\Sigma_2]\!] \to ((x : [\![T_2]\!]) \to [\![C_{21}]\!]) \to [\![C_{22}]\!]$.

By inversion, we have

- $\Gamma, \beta \vdash \tau$,
- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!][\tau/\alpha]$,
- $\Gamma, \beta, h : [\![\Sigma_2]\!] \vdash [\![T_1]\!][\tau/\alpha] <: [\![T_2]\!]$,
- $\Gamma, \beta, h : [\![\Sigma_2]\!], x : [\![T_1]\!][\tau/\alpha] \vdash [\![C_{21}]\!] <: [\![C_{11}]\!][\tau/\alpha]$, and
- $\Gamma, \beta, h : [\![\Sigma_2]\!], k : (x : [\![T_2]\!]) \to [\![C_{21}]\!] \vdash [\![C_{12}]\!][\tau/\alpha] <: [\![C_2 2]\!]$.

for some τ. Since CPS-transformed types do not contain type variables, we have

- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!]$,
- $\Gamma, \beta, h : [\![\Sigma_2]\!] \vdash [\![T_1]\!] <: [\![T_2]\!]$,
- $\Gamma, \beta, h : [\![\Sigma_2]\!], x : [\![T_1]\!] \vdash [\![C_{21}]\!] <: [\![C_{11}]\!]$, and
- $\Gamma, \beta, h : [\![\Sigma_2]\!], k : (x : [\![T_2]\!]) \to [\![C_{21}]\!] \vdash [\![C_{12}]\!] <: [\![C_2 2]\!]$.

By 9, we have

- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!]$,
- $\Gamma, \beta \vdash [\![T_1]\!] <: [\![T_2]\!]$,
- $\Gamma, \beta, x : [\![T_1]\!] \vdash [\![C_{21}]\!] <: [\![C_{11}]\!]$, and
- $\Gamma, \beta \vdash [\![C_{12}]\!] <: [\![C_2 2]\!]$.

By the IHs, we have

- $(\!|\Gamma|\!) \vdash \Sigma_2 <: \Sigma_1$,
- $(\!|\Gamma|\!) \vdash T_1 <: T_2$,
- $(\!|\Gamma|\!), x : T_1 \vdash C_{21} <: C_{11}$, and
- $(\!|\Gamma|\!) \vdash C_{12} <: C_{22}$.

Then we have the conclusion by (S-ATM) and (S-Comp).

Case $C_1 = \Sigma_1 \triangleright T_1 \ / \ \square$ **and** $C_2 = \Sigma_2 \triangleright T_2 \ / \ (\forall x.C_{21}) \Rightarrow C_{22}$**:** We have

- $[\![C_1]\!] = \forall \alpha.[\![\Sigma_1]\!] \to ([\![T_1]\!] \to \alpha) \to \alpha$ and
- $[\![C_2]\!] = \forall \beta.[\![\Sigma_2]\!] \to ((x : [\![T_2]\!]) \to [\![C_{21}]\!]) \to [\![C_{22}]\!]$.

W.l.o.g., we can assume that $x \notin [\![C_{22}]\!]$. By inversion, we have

- $\Gamma, \beta \vdash \tau$,
- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!][\tau/\alpha]$,
- $\Gamma, \beta, h : [\![\Sigma_2]\!] \vdash [\![T_1]\!][\tau/\alpha] <: [\![T_2]\!]$,
- $\Gamma, \beta, h : [\![\Sigma_2]\!], x : [\![T_1]\!][\tau/\alpha] \vdash [\![C_{21}]\!] <: \alpha[\tau/\alpha]$, and
- $\Gamma, \beta, h : [\![\Sigma_2]\!], k : (x : [\![T_2]\!]) \to [\![C_{21}]\!] \vdash \alpha[\tau/\alpha] <: [\![C_2 2]\!]$.

for some τ. Since CPS-transformed types do not contain type variables, we have

- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!]$,
- $\Gamma, \beta, h : [\![\Sigma_2]\!] \vdash [\![T_1]\!] <: [\![T_2]\!]$,
- $\Gamma, \beta, h : [\![\Sigma_2]\!], x : [\![T_1]\!] \vdash [\![C_{21}]\!] <: \tau$, and
- $\Gamma, \beta, h : [\![\Sigma_2]\!], k : (x : [\![T_2]\!]) \to [\![C_{21}]\!] \vdash \tau <: [\![C_2 2]\!]$.

By 9, we have

- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!]$,
- $\Gamma, \beta \vdash [\![T_1]\!] <: [\![T_2]\!]$,
- $\Gamma, \beta, x : [\![T_1]\!] \vdash [\![C_{21}]\!] <: \tau$, and
- $\Gamma, \beta \vdash \tau <: [\![C_2 2]\!]$.

By Lemma 35 and 47, we have

- $\Gamma, \beta \vdash [\![\Sigma_2]\!] <: [\![\Sigma_1]\!]$,
- $\Gamma, \beta \vdash [\![T_1]\!] <: [\![T_2]\!]$, and
- $\Gamma, \beta, x : [\![T_1]\!] \vdash [\![C_{21}]\!] <: [\![C_2 2]\!]$.

By the IHs, we have

- $(\!|\Gamma|\!) \vdash \Sigma_2 <: \Sigma_1$,
- $(\!|\Gamma|\!) \vdash T_1 <: T_2$, and
- $(\!|\Gamma|\!), x : T_1 \vdash C_{21} <: C_{22}$.

Then we have the conclusion by (S-Embed) and (S-Comp).

Case $C_1 = \Sigma_1 \triangleright T_1 \ / \ (\forall x.C_{11}) \Rightarrow C_{12}$ **and** $C_2 = \Sigma_2 \triangleright T_2 \ / \ \square$**:** We have

- $[\![C_1]\!] = \forall\alpha.[\![\Sigma_1]\!] \to ((x : [\![T_1]\!]) \to [\![C_{11}]\!]) \to [\![C_{12}]\!]$ and
- $[\![C_2]\!] = \forall\beta.[\![\Sigma_2]\!] \to ([\![T_2]\!] \to \beta) \to \beta$.

By inversion, we have

- $\Gamma, \beta \vdash \tau$, and
- $\Gamma, \beta, h : [\![\Sigma_2]\!], k : (x : [\![T_2]\!]) \to \beta \vdash [\![C_{12}]\!][\tau/\alpha] <: \beta$.

for some τ. Since CPS-transformed types do not contain type variables, we have

- $\Gamma, \beta, h : [\![\Sigma_2]\!], k : (x : [\![T_2]\!]) \to \beta \vdash [\![C_{12}]\!] <: \beta$.

This is contradictory since $[\![C_{12}]\!]$ cannot be a type variable and thus there is no applicable rule.

3. By the IHs, and (S-Sig).

□

Theorem 67 (Backward type preservation (for open expressions))**.** *Assume that* Γ *is cps-wellformed.*

1. If $\Gamma \vdash [\![v]\!] : \tau$*, then there exists* T *such that* $(\!|\Gamma|\!) \vdash v : T$ *and* $\Gamma \vdash [\![T]\!] <: \tau$*.*

2. If $\Gamma \vdash [\![c]\!] : \tau$*, then there exists* C *such that* $(\!|\Gamma|\!) \vdash c : C$ *and* $\Gamma \vdash [\![C]\!] <: \tau$*.*

Proof. By simultaneous induction on the structure of v and c.

1. **Case** $v = x$**:** We have $[\![v]\!] = x$. By Lemma 48, we have *either*

1. $\vdash \Gamma$ and $\Gamma \vdash \{z : B \mid z = x\} <: \tau$ (if $\Gamma(x) = \{z : B \mid \phi\}$ for some z, B and ϕ)
2. $\vdash \Gamma$ and $\Gamma \vdash \Gamma(x) <: \tau$ (otherwise)

Case 1: By Lemma 65, we have $\vdash (\!|\Gamma|\!)$. Also, since $[\![\{z : B \mid \phi'\}]\!] = \{z : B \mid \phi'\}$ for any ϕ', we have

- $(\!|\Gamma|\!)(x) = \{z : B \mid \phi\}$ and
- $\Gamma \vdash [\![\{z : B \mid z = x\}]\!] <: \tau$.

Then, by (T-CVar), we have $(\!|\Gamma|\!) \vdash x : \{z : B \mid z = x\}$. Now we have the conclusion with $T = \{z : B \mid z = x\}$.

Case 2: By Lemma 65, we have $\vdash (\!|\Gamma|\!)$. Then, since $\Gamma(x) = rmtv(\Gamma)(x) = [\![(\!|\Gamma|\!)]\!](x) = [\![(\!|\Gamma|\!)(x)]\!]$ holds by Lemma 60, we have $\Gamma \vdash [\![(\!|\Gamma|\!)(x)]\!] <: \tau$. Also, by (T-Var), we have $(\!|\Gamma|\!) \vdash x : (\!|\Gamma|\!)(x)$. Now we have the conclusion with $T = (\!|\Gamma|\!)(x)$.

Case $v = p$**:** We have $[\![v]\!] = cps(p)$. By Lemma 48, we have

- $\vdash \Gamma$ and
- $\Gamma \vdash ty_{cps}([\![p]\!]) <: \tau$.

By Lemma 65, we have $\vdash (\!|\Gamma|\!)$. Then, by (T-Prim), we have $(\!|\Gamma|\!) \vdash p : ty(p)$. Also, by Assumption 52, we have $\Gamma \vdash [\![ty(p)]\!] <: \tau$. Now we have the conclusion with $T = ty(p)$.

Case $v = \mathbf{rec}(f^{(x:T_1)\to C_1}, x^{T_2}).c$**:** We have $[\![v]\!] = \mathbf{rec}(f : (x : [\![T_1]\!]) \to [\![C_1]\!], x : [\![T_2]\!]).[\![c]\!]$. By Lemma 48, we have

- $\Gamma, f : (x : [\![T_1]\!]) \to [\![C_1]\!], x : [\![T_1]\!] \vdash [\![c]\!] : [\![C_1]\!]$ and
- $\Gamma \vdash (x : [\![T_1]\!]) \to [\![C_1]\!] <: \tau$.

By the IH, we have

- $(\!|\Gamma|\!), f : (x : T_1) \to C_1, x : T_1 \vdash c : C_1'$ and
- $\Gamma, f : (x : [\![T_1]\!]) \to [\![C_1]\!], x : [\![T_1]\!] \vdash [\![C_1']\!] <: [\![C_1]\!]$

for some C_1'. By Lemma 66, we have

$$(\!|\Gamma|\!), f : (x : T_1) \to C_1, x : T_1 \vdash C_1' <: C_1 \ .$$

Then, by (T-CSub) and (T-Fun), we have

$$\Gamma \vdash \mathbf{rec}(f^{(x:T_1) \to C_1}, x^{T_2}).c : (x : T_1) \to C_1 \ .$$

Now we have the conclusion with $T = (x : T_1) \to C_1$.

2. **Case $c = \mathbf{return}\ v^T$:** We have $[\![c]\!] = \Lambda\alpha.\lambda h : \{\}.\lambda k : [\![T]\!] \to \alpha.k\ [\![v]\!]$. By Lemma 49, we have

- $\Gamma, \alpha, h : \{\}, k : [\![T]\!] \to \alpha \vdash k\ [\![v]\!] : \tau'$ and
- $\Gamma \vdash \forall\alpha.\{\} \to ([\![T]\!] \to \alpha) \to \tau' <: \tau$

for some τ'. By Lemma 48, we have

- $\Gamma, \alpha, h : \{\}, k : [\![T]\!] \to \alpha \vdash k : (y : \tau_1) \to \tau_2$,
- $\Gamma, \alpha, h : \{\}, k : [\![T]\!] \to \alpha \vdash [\![v]\!] : \tau_1$, and
- $\Gamma, \alpha, h : \{\}, k : [\![T]\!] \to \alpha \vdash \tau_2[[\![v]\!]/y] <: \tau'$

for some y, τ_1, and τ_2. By Lemma 39, we have

- $\Gamma, \alpha \vdash [\![v]\!] : \tau_1$.

Then, by the IH, we have

- $(\!|\Gamma|\!) \vdash v : T$ and
- $\Gamma \vdash [\![T]\!] <: \tau_1$.

Therefore, by (T-Ret), we have

- $(\!|\Gamma|\!) \vdash \mathbf{return}\ v : \{\} \triangleright T \ /\ \square$.

On the other hand, by Lemma 48, we have

- $\Gamma, \alpha, h : \{\}, k : [\![T]\!] \to \alpha \vdash [\![T]\!] \to \alpha <: (y : \tau_1) \to \tau_2$.

Then, By inversion, we have $\tau_2 = \alpha$. By inversion again, we have $\tau' = \alpha$. Therefore, we have

- $\Gamma \vdash \forall\alpha.\{\} \to ([\![T]\!] \to \alpha) \to \alpha <: \tau$,

that is,

- $\Gamma \vdash [\![\{\} \triangleright T \ /\ \square]\!] <: \tau$.

Now we have the conclusion with $C = \{\} \triangleright T \ /\ \square$.

Case $c = \mathbf{let}\ x = c_1^{\Sigma \triangleright T_1 / \square}\ \mathbf{in}\ c_2^{\Sigma \triangleright T_2 / \square}$: We have $[\![c]\!] = \Lambda\alpha.\lambda h : [\![\Sigma]\!].\lambda k : [\![T_2]\!] \to \alpha.[\![c_1]\!]\ \alpha\ h\ (\lambda x : [\![T_1]\!].[\![c_2]\!]\ \alpha\ h\ k)$. By Lemma 49, we have

(i) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha \vdash [\![c_1]\!]\ \alpha\ h\ (\lambda x : [\![T_1]\!].[\![c_2]\!]\ \alpha\ h\ k) : \tau'$ and

(ii) $\Gamma \vdash \forall\alpha.[\![\Sigma]\!] \to ([\![T_2]\!] \to \alpha) \to \tau' <: \tau$

for some τ'. By Lemma 50 with (i), we have

(iii) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha \vdash [\![c_1]\!] : \tau''$,

(iv) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha \vdash h : \tau_1$, and

(v) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha \vdash \lambda x : [\![T_1]\!].[\![c_2]\!]\ \alpha\ h\ k : \tau_2$

for some τ'', τ_1 and τ_2. By Lemma 48 with (iv) and (v) respectively, we have

(vi) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha \vdash [\![\Sigma]\!] <: \tau_1$,

(vii) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha, x : [\![T_1]\!] \vdash [\![c_2]\!]\ \alpha\ h\ k : \tau_3$, and

(viii) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha \vdash (x : [\![T_1]\!]) \to \tau_3 <: \tau_2$

for some τ_3. Then, by the second half of Lemma 50, we have

(ix) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha \vdash \tau'' <: \forall\beta.[\![\Sigma]\!] \to ((x : [\![T_1]\!]) \to \tau_3) \to \tau'$

where β is fresh.

On the other hand, by Lemma 50 with (vii), we have

(x) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha, x : [\![T_1]\!] \vdash [\![c_2]\!] : \tau_3'$,

(xi) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha, x : [\![T_1]\!] \vdash h : \tau_4$, and

(xii) $\Gamma, \alpha, h : [\![\Sigma]\!], k : [\![T_2]\!] \to \alpha, x : [\![T_1]\!] \vdash k : \tau_5$

for some τ_3', τ_4 and τ_5. By Lemma 48 with (xi) and (xii) respectively, we have

- $\Gamma, \alpha, h : \llbracket\Sigma\rrbracket, k : \llbracket T_2\rrbracket \to \alpha, x : \llbracket T_1\rrbracket \vdash \llbracket\Sigma\rrbracket <: \tau_4$ and
- $\Gamma, \alpha, h : \llbracket\Sigma\rrbracket, k : \llbracket T_2\rrbracket \to \alpha, x : \llbracket T_1\rrbracket \vdash \llbracket T_2\rrbracket \to \alpha <: \tau_5$.

Then, by the second half of Lemma 50, we have

(xiii) $\Gamma, \alpha, h : \llbracket\Sigma\rrbracket, k : \llbracket T_2\rrbracket \to \alpha, x : \llbracket T_1\rrbracket \vdash \tau_3' <: \forall\gamma.\llbracket\Sigma\rrbracket \to (\llbracket T_2\rrbracket \to \alpha) \to \tau_3$

where γ is fresh.

By Lemma 39 with (iii), (ix), (x) and (xiii), we have

(xiv) $\Gamma, \alpha \vdash \llbracket c_1\rrbracket : \tau''$,

(xv) $\Gamma, \alpha \vdash \tau'' <: \forall\beta.\llbracket\Sigma\rrbracket \to ((x : \llbracket T_1\rrbracket) \to \tau_3) \to \tau'$

(xvi) $\Gamma, \alpha, x : \llbracket T_1\rrbracket \vdash \llbracket c_2\rrbracket : \tau_3'$

(xvii) $\Gamma, \alpha, x : \llbracket T_1\rrbracket \vdash \tau_3' <: \forall\gamma.\llbracket\Sigma\rrbracket \to (\llbracket T_2\rrbracket \to \alpha) \to \tau_3$.

Then, by the IHs of (xiv) and (xvi) respectively, we have

(xviii) $(\!|\Gamma|\!) \vdash c_1 : C_1$,

(xix) $\Gamma, \alpha \vdash \llbracket C_1\rrbracket <: \tau''$,

(xx) $(\!|\Gamma|\!), x : T_1 \vdash c_2 : C_2$, and

(xxi) $\Gamma, \alpha, x : \llbracket T_1\rrbracket \vdash \llbracket C_2\rrbracket <: \tau_3'$

for some C_1 and C_2. By Lemma 47 with "(xv) and (xix)" and "(xvii) and (xxi)" respectively, we have

(xxii) $\Gamma, \alpha \vdash \llbracket C_1\rrbracket <: \forall\beta.\llbracket\Sigma\rrbracket \to ((x : \llbracket T_1\rrbracket) \to \tau_3) \to \tau'$ and

(xxiii) $\Gamma, \alpha, x : \llbracket T_1\rrbracket \vdash \llbracket C_2\rrbracket <: \forall\gamma.\llbracket\Sigma\rrbracket \to (\llbracket T_2\rrbracket \to \alpha) \to \tau_3$.

By Lemma 63 with (xxiii), we have

- $C_1 = \Sigma_{11} \triangleright T_{11} \ / \ \square$ and
- $\tau_3 = \alpha$

for some Σ_{11} and T_{11}. Then, by Lemma 63 again with (xxii), we have

- $C_2 = \Sigma_{22} \triangleright T_{22} \ / \ \square$ and
- $\tau' = \alpha$

for some Σ_{22} and T_{22}. By inversion of (xxii), we have

- $\Gamma, \alpha, \beta \vdash \llbracket\Sigma\rrbracket <: \llbracket\Sigma_{11}\rrbracket$ and
- $\Gamma, \alpha, h : \llbracket\Sigma\rrbracket, \beta \vdash \llbracket T_{11}\rrbracket <: \llbracket T_1\rrbracket$.

By Lemma 39, 40 and 66, we have

- $(\!|\Gamma|\!) \vdash \Sigma <: \Sigma_{11}$ and
- $(\!|\Gamma|\!) \vdash T_{11} <: T_1$.

Then, by subsumption on (xviii), we have

(xxiv) $(\!|\Gamma|\!) \vdash c_1 : \Sigma \triangleright T_1 \ / \ \square$.

In the same way, from (xxiii), we have

(xxv) $(\!|\Gamma|\!), x : T_1 \vdash c_2 : \Sigma \triangleright T_2 \ / \ \square$.

Therefore, by (T-LetP) with (xxiv) and (xxv), we have

$$(\!|\Gamma|\!) \vdash \textbf{let } x = c_1 \textbf{ in } c_2 : \Sigma \triangleright T_2 \ / \ \square .$$

Also, since $\tau' = \alpha$, (ii) implies

$$\Gamma \vdash \llbracket\Sigma \triangleright T_2 \ / \ \square\rrbracket <: \tau .$$

Now we have the conclusion with $C = \Sigma \triangleright T_2 \ / \ \square$.

Case $c = \textbf{let } x = c_1^{\Sigma \triangleright T_1 / (\forall x.C_1) \Rightarrow C_2} \textbf{ in } c_2^{\Sigma \triangleright T_2 / (\forall z.C_0) \Rightarrow C_1}$**:** We have $\llbracket c\rrbracket = \Lambda\alpha.\lambda h : \llbracket\Sigma\rrbracket.\lambda k : (z : \llbracket T_2\rrbracket) \to \llbracket C_0\rrbracket.\llbracket c_1\rrbracket \ \llbracket C_2\rrbracket \ h \ (\lambda x : \llbracket T_1\rrbracket.\llbracket c_2\rrbracket \ \llbracket C_1\rrbracket \ h \ k)$. In the similar way to the previous case, we have

(i) $\Gamma \vdash \forall\alpha.\llbracket\Sigma\rrbracket \to ((z : \llbracket T_2\rrbracket) \to \llbracket C_0\rrbracket) \to \tau' <: \tau$,

(ii) $(\!|\Gamma|\!) \vdash c_1 : C_1$,

(iii) $\Gamma, \alpha \vdash \llbracket C_1\rrbracket <: \forall\beta.\llbracket\Sigma\rrbracket \to ((x : \llbracket T_1\rrbracket) \to \tau_3) \to \tau'$,

(iv) $(\!|\Gamma|\!), x : T_1 \vdash c_2 : C_2$, and

(v) $\Gamma, \alpha, x : \llbracket T_1\rrbracket \vdash \llbracket C_2\rrbracket <: \forall\gamma.\llbracket\Sigma\rrbracket \to ((z : \llbracket T_2\rrbracket) \to \llbracket C_0\rrbracket) \to \tau_3$

for some τ', τ_3, C_1 and C_2. By Lemma 64, we can assume that

- $C_1 = \Sigma_1 \triangleright T_{10} \ / \ (\forall x_1.C_{11}) \Rightarrow C_{12}$ and
- $C_2 = \Sigma_2 \triangleright T_{20} \ / \ (\forall x_2.C_{21}) \Rightarrow C_{22}$

for some $\Sigma_1, T_{10}, x_1, C_{11}, C_{12}, \Sigma_2, T_{20}, x_2, C_{21}$ and C_{22}. Then, by inversion of (iii), we have

(vi) $x_1 = x$

(vii) $\Gamma, \alpha, \beta \vdash [\![\Sigma]\!] <: [\![\Sigma_1]\!]$,

(viii) $\Gamma, \alpha, \beta, h : [\![\Sigma]\!] \vdash [\![T_{10}]\!] <: [\![T_1]\!]$,

(ix) $\Gamma, \alpha, \beta, h : [\![\Sigma]\!], x : [\![T_{10}]\!] \vdash \tau_3 <: [\![C_{11}]\!]$, and

(x) $\Gamma, \alpha, \beta, h : [\![\Sigma]\!], k : (x : [\![T_1]\!]) \to \tau_3 \vdash [\![C_{12}]\!] <: \tau'$.

By Lemma 39, 40 and 66 with (vii) and (viii) respectively, we have

(xi) $(\!|\Gamma|\!) \vdash \Sigma <: \Sigma_1$ and

(xii) $(\!|\Gamma|\!) \vdash T_{10} <: T_1$.

By subsumption on (ii) with (xi), we have

(xiii) $(\!|\Gamma|\!) \vdash c_1 : \Sigma \triangleright T_{10} \ / \ (\forall x_1.C_{11}) \Rightarrow C_{12}$.

On the other hand, by inversion of (v), we have

- $x_2 = z$
- $\Gamma, \alpha, x : [\![T_1]\!], \gamma \vdash [\![\Sigma]\!] <: [\![\Sigma_2]\!]$,
- $\Gamma, \alpha, x : [\![T_1]\!], \gamma, h : [\![\Sigma]\!] \vdash [\![T_{20}]\!] <: [\![T_2]\!]$,
- $\Gamma, \alpha, x : [\![T_1]\!], \gamma, h : [\![\Sigma]\!], z : [\![T_{20}]\!] \vdash [\![C_0]\!] <: [\![C_{21}]\!]$, and
- $\Gamma, \alpha, x : [\![T_1]\!], \gamma, h : [\![\Sigma]\!], k : (z : [\![T_2]\!]) \to [\![C_0]\!] \vdash [\![C_{22}]\!] <: \tau_3$.

By Lemma 36 with (viii), we have

(xiv) $\Gamma, \alpha, x : [\![T_{10}]\!], \gamma \vdash [\![\Sigma]\!] <: [\![\Sigma_2]\!]$,

(xv) $\Gamma, \alpha, x : [\![T_{10}]\!], \gamma, h : [\![\Sigma]\!] \vdash [\![T_{20}]\!] <: [\![T_2]\!]$,

(xvi) $\Gamma, \alpha, x : [\![T_{10}]\!], \gamma, h : [\![\Sigma]\!], z : [\![T_{20}]\!] \vdash [\![C_0]\!] <: [\![C_{21}]\!]$, and

(xvii) $\Gamma, \alpha, x : [\![T_{10}]\!], \gamma, h : [\![\Sigma]\!], k : (z : [\![T_2]\!]) \to [\![C_0]\!] \vdash [\![C_{22}]\!] <: \tau_3$.

By Lemma 39, 40 and 47 with (ix) and (xvii), we have

(xviii) $\Gamma, \alpha, x : [\![T_{10}]\!] \vdash [\![C_{22}]\!] <: [\![C_{11}]\!]$.

By Lemma 39, 40 and 66 with (xiv), (xv), (xvi) and (xviii), we have

- $(\!|\Gamma|\!), x : T_{10} \vdash \Sigma <: \Sigma_2$,
- $(\!|\Gamma|\!), x : T_{10} \vdash T_{20} <: T_2$,
- $(\!|\Gamma|\!), x : T_{10}, z : T_{20} \vdash C_0 <: C_{21}$, and
- $(\!|\Gamma|\!), x : T_{10} \vdash C_{22} <: C_{11}$.

Then, by Lemma 36 and subsumption on (iv), we have

(xix) $(\!|\Gamma|\!), x : T_{10} \vdash c_2 : \Sigma \triangleright T_2 \ / \ (\forall z.C_0) \Rightarrow C_{11}$.

Therefore, by (T-LetIp), we have

$$(\!|\Gamma|\!) \vdash \textbf{let } x = c_1 \textbf{ in } c_2 : \Sigma \triangleright T_2 \ / \ (\forall z.C_0) \Rightarrow C_{12} .$$

Also, by Lemma 39, 40, 35, and 47 with (i) and (x), we have

- $\Gamma \vdash \forall \alpha.[\![\Sigma]\!] \to ((z : [\![T_2]\!]) \to [\![C_0]\!]) \to [\![C_{12}]\!] <: \tau$,

that is,

$$\Gamma \vdash [\![\Sigma \triangleright T_2 \ / \ (\forall z.C_0) \Rightarrow C_{12}]\!] <: \tau .$$

Now we have the conclusion with $C = \Sigma \triangleright T_2 \ / \ (\forall z.C_0) \Rightarrow C_{12}$.

Case $c = v_1\ v_2$**:** We have $[\![c]\!] = [\![v_1]\!]\ [\![v_2]\!]$. By Lemma 48, we have

- $\Gamma \vdash [\![v_1]\!] : (x : \tau_1) \to \tau_2$,
- $\Gamma \vdash [\![v_2]\!] : \tau_1$, and
- $\Gamma \vdash \tau_2[[\![v_2]\!]/x] <: \tau$

for some x, τ_1 and τ_2. By the IHs, we have

- $(\!|\Gamma|\!) \vdash v_1 : T_1$,
- $(\!|\Gamma|\!) \vdash v_2 : T_2$,
- $\Gamma \vdash [\![T_1]\!] <: (x : \tau_1) \to \tau_2$, and

- $\Gamma \vdash [\![T_2]\!] <: \tau_1$

for some T_1 and T_2. By inversion, we have

- $T_1 = (x : T_{11}) \rightarrow C_{12}$,
- $\Gamma \vdash \tau_1 <: [\![T_{11}]\!]$, and
- $\Gamma, x : \tau_1 \vdash [\![C_{12}]\!] <: \tau_2$

for some T_{11} and C_{12}. By Lemma 47, we have $\Gamma \vdash [\![T_2]\!] <: [\![T_{11}]\!]$. Then, by Lemma 66, we have $(\!|\Gamma|\!) \vdash T_2 <: T_{11}$, and hence by (T-VSub) we have $(\!|\Gamma|\!) \vdash v_2 : T_{11}$. Therefore, by (T-App), we have $(\!|\Gamma|\!) \vdash v_1\ v2 : C_{12}[v_2/x]$.

On the other hand, by Lemma 41, we have $\Gamma \vdash [\![C_{12}]\!][[\![v_2]\!]/x] <: \tau_2[[\![v_2]\!]/x]$. Then, by Lemma 47, we have $\Gamma \vdash [\![C_{12}]\!][[\![v_2]\!]/x] <: \tau$.

Now we have the conclusion with $C = C_{12}[v_2/x]$.

Case $c = (\mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2)^{C'}$**:** We have $[\![c]\!] = (\mathbf{if}\ [\![v]\!]\ \mathbf{then}\ [\![c_1]\!]\ \mathbf{else}\ [\![c_2]\!] : [\![C']\!])$. By Lemma 48, we have

- $\Gamma \vdash \mathbf{if}\ [\![v]\!]\ \mathbf{then}\ [\![c_1]\!]\ \mathbf{else}\ [\![c_2]\!] : [\![C']\!]$ and
- $\Gamma \vdash [\![C']\!] <: \tau$.

By Lemma 48 again, we have

- $\Gamma \vdash [\![v]\!] : \{z : \mathrm{bool} \mid \phi\}$,
- $\Gamma, [\![v]\!] = \mathbf{true} \vdash [\![c_1]\!] : \tau'$,
- $\Gamma, [\![v]\!] = \mathbf{false} \vdash [\![c_2]\!] : \tau'$, and
- $\Gamma \vdash \tau' <: [\![C']\!]$

for some z, ϕ and τ'. By the IHs, we have

- $(\!|\Gamma|\!) \vdash v : \{z : \mathrm{bool} \mid \phi\}$,
- $(\!|\Gamma|\!), v = \mathbf{true} \vdash c_1 : C_1$,
- $(\!|\Gamma|\!), v = \mathbf{false} \vdash c_2 : C_2$,
- $\Gamma, [\![v]\!] = \mathbf{true} \vdash [\![C_1]\!] <: \tau'$, and
- $\Gamma, [\![v]\!] = \mathbf{false} \vdash [\![C_2]\!] <: \tau'$

for some C_1 and C_2. (Note that since v is of a refinement type, it holds that $[\![v]\!] = v$.) By Lemma 35 and 47, we have

- $\Gamma, [\![v]\!] = \mathbf{true} \vdash [\![C_1]\!] <: [\![C']\!]$ and
- $\Gamma, [\![v]\!] = \mathbf{false} \vdash [\![C_2]\!] <: [\![C']\!]$.

By Lemma 66, we have

- $(\!|\Gamma|\!), v = \mathbf{true} \vdash C_1 <: C'$ and
- $(\!|\Gamma|\!), v = \mathbf{false} \vdash C_2 <: C'$.

Then, by (T-CSub), we have

- $(\!|\Gamma|\!), v = \mathbf{true} \vdash c_1 : C'$ and
- $(\!|\Gamma|\!), v = \mathbf{false} \vdash c_2 : C'$.

Therefore by (T-If), we have $(\!|\Gamma|\!) \vdash \mathbf{if}\ v\ \mathbf{then}\ c_1\ \mathbf{else}\ c_2 : C'$. Now we have the conclusion with $C = C'$.

Case $c = (\mathsf{op}^{\widetilde{A}}\ v)^{\Sigma \triangleright T/(\forall y.C_1) \Rightarrow C_2}$**:** We have $[\![c]\!] = \Lambda\alpha.\lambda h : [\![\Sigma]\!].\lambda k : (y : [\![T]\!]) \rightarrow [\![C_1]\!].h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y' : [\![T]\!].k\ y')$. By Lemma 49, we have

(i) $\Gamma, \alpha, h : [\![\Sigma]\!], k : (y : [\![T]\!]) \rightarrow [\![C_1]\!] \vdash h\#\mathsf{op}\ \widetilde{A}\ [\![v]\!]\ (\lambda y' : [\![T]\!].k\ y') : \tau'$ and
(ii) $\Gamma \vdash \forall\alpha.[\![\Sigma]\!] \rightarrow ((y : [\![T]\!]) \rightarrow [\![C_1]\!]) \rightarrow \tau' <: \tau$

for some τ'. (Below, we write $\Gamma_{\alpha,h,k}$ for $\Gamma, \alpha, h : [\![\Sigma]\!], k : (y : [\![T]\!]) \rightarrow [\![C_1]\!]$.)

By Lemma 48 with (i), we have

(iii) $\Gamma_{\alpha,h,k} \vdash \lambda y' : [\![T]\!].k\ y' : \tau_1$,
(iv) $\Gamma_{\alpha,h,k} \vdash [\![v]\!] : \tau_3$,
(v) $\Gamma_{\alpha,h,k} \vdash \widetilde{A : \widetilde{B}}$,
(vi) $\Gamma_{\alpha,h,k} \vdash [\![\Sigma]\!] <: \{\ldots, \forall \widetilde{X : \widetilde{B}}.\tau_5, \ldots\}$,
(vii) $\Gamma_{\alpha,h,k} \vdash \tau_5[\widetilde{A/X}] <: (x : \tau_3) \rightarrow \tau_4$,
(viii) $\Gamma_{\alpha,h,k} \vdash \tau_4[[\![v]\!]/x] <: \tau_1 \rightarrow \tau_2$, and

(ix) $\Gamma_{\alpha,h,k} \vdash \tau_2 <: \tau'$.

By Assumption 33 and 62 with (v), we have

- $(\!|\Gamma|\!) \vdash \widetilde{A : \widetilde{B}}$.

By inversion of (vi), we have

- $\Sigma = \{\ldots, \mathsf{op} : \forall \widetilde{X : \widetilde{B}}.(x_{\mathsf{op}} : T_{\mathsf{op1}}) \rightarrow ((y_{\mathsf{op}} : T_{\mathsf{op2}}) \rightarrow C_{\mathsf{op1}}) \rightarrow C_{\mathsf{op2}}, \ldots\}$ and
- $\Gamma_{\alpha,h,k}, \widetilde{X : \widetilde{B}} \vdash (x_{\mathsf{op}} : [\![T_{\mathsf{op1}}]\!]) \rightarrow ((y_{\mathsf{op}} : [\![T_{\mathsf{op2}}]\!]) \rightarrow [\![C_{\mathsf{op1}}]\!]) \rightarrow [\![C_{\mathsf{op2}}]\!] <: \tau_5$.

By repeatedly inverting this subtyping judgment with applying Lemma 42 with (v), Lemma 41 with (iv), and Lemma 47 with (vii), (viii) and (ix), we have

(x) $x = x_{\mathsf{op}}$,

(xi) $\Gamma_{\alpha,h,k} \vdash \tau_3 <: [\![T_{\mathsf{op1}}]\!][\widetilde{A/X}]$,

(xii) $\Gamma_{\alpha,h,k} \vdash \tau_1 <: (y_{\mathsf{op}} : [\![T_{\mathsf{op2}}]\!][\widetilde{A/X}][[\![v]\!]/x]) \rightarrow [\![C_{\mathsf{op1}}]\!][\widetilde{A/X}][[\![v]\!]/x]$, and

(xiii) $\Gamma_{\alpha,h,k} \vdash [\![C_{\mathsf{op2}}]\!][\widetilde{A/X}][[\![v]\!]/x] <: \tau'$.

By Lemma 39 with (iv), we have

- $\Gamma, \alpha \vdash [\![v]\!] : \tau_3$.

Then, by the IH, we have

(xiv) $(\!|\Gamma|\!) \vdash v : T_v$ and

(xv) $\Gamma, \alpha \vdash [\![T_v]\!] <: \tau_3$

for some T_v. By Lemma 47 with (xi) and (xv) (using Lemma 39), we have

- $\Gamma, \alpha \vdash [\![T_v]\!] <: [\![T_{\mathsf{op1}}]\!][\widetilde{A/X}]$.

Then, by Lemma 66, we have

- $(\!|\Gamma|\!) \vdash T_v <: T_{\mathsf{op1}}[\widetilde{A/X}]$

and hence, by (T-VSub) with (xiv), we have

- $(\!|\Gamma|\!) \vdash v : T_{\mathsf{op1}}[\widetilde{A/X}]$.

Also, by Lemma 44 and inversion, we have $\Gamma, \alpha \vdash [\![\Sigma]\!]$. Then by Lemma 65, we have $(\!|\Gamma|\!) \vdash \Sigma$. Therefore, by (T-Op), we have

$$(\!|\Gamma|\!) \vdash \mathsf{op}\ v : \Sigma \triangleright T_{\mathsf{op2}}[\widetilde{A/X}][v/x] \ / \ (\forall y_{\mathsf{op}}.C_{\mathsf{op1}}[\widetilde{A/X}][v/x]) \Rightarrow C_{\mathsf{op2}}[\widetilde{A/X}][v/x] \ .$$

On the other hand, by Lemma 48 with (iii), we have

(xvi) $\Gamma_{\alpha,h,k} \vdash (y' : [\![T]\!]) \rightarrow \tau_6 <: \tau_1$,

(xvii) $\Gamma_{\alpha,h,k}, y' : [\![T]\!] \vdash \tau_8[y'/y_0] <: \tau_6$,

(xviii) $\Gamma_{\alpha,h,k}, y' : [\![T]\!] \vdash (y : [\![T]\!]) \rightarrow [\![C_1]\!] <: (y_0 : \tau_7) \rightarrow \tau_8$, and

(xix) $\Gamma_{\alpha,h,k}, y' : [\![T]\!] \vdash y' : \tau_7$.

By inversion of (xviii), we have

- $y = y_0$ and
- $\Gamma_{\alpha,h,k}, y' : [\![T]\!], y : \tau_7 \vdash [\![C_1]\!] <: \tau_8$.

Then, by Lemma 41 with (xix), we have

- $\Gamma_{\alpha,h,k}, y' : [\![T]\!] \vdash [\![C_1]\!][y'/y] <: \tau_8[y'/y]$.

Then, by Lemma 47 with (xvii), we have

- $\Gamma_{\alpha,h,k}, y' : [\![T]\!] \vdash [\![C_1]\!][y'/y] <: \tau_6$

(Note that $y = y_0$). Then by (Sc-Fun), we have

- $\Gamma_{\alpha,h,k} \vdash (y' : [\![T]\!]) \rightarrow [\![C_1]\!][y'/y] <: (y' : [\![T]\!]) \rightarrow \tau_6$

and by α-renaming we have

- $\Gamma_{\alpha,h,k} \vdash (y : [\![T]\!]) \rightarrow [\![C_1]\!] <: (y' : [\![T]\!]) \rightarrow \tau_6$.

Then, by Lemma 47 with (xvi) and (xii), we have

(xx) $\Gamma_{\alpha,h,k} \vdash (y : [\![T]\!]) \rightarrow [\![C_1]\!] <: (y_{\mathsf{op}} : [\![T_{\mathsf{op2}}]\!][\widetilde{A/X}][[\![v]\!]/x]) \rightarrow [\![C_{\mathsf{op1}}]\!][\widetilde{A/X}][[\![v]\!]/x]$.

Therefore, by some subtyping rules with (xx) and (xiii), we have

- $\Gamma \vdash \forall\alpha.[\![\Sigma]\!] \rightarrow ((y_{\mathsf{op}} : [\![T_{\mathsf{op2}}]\!][\widetilde{A/X}][[\![v]\!]/x]) \rightarrow [\![C_{\mathsf{op1}}]\!][\widetilde{A/X}][[\![v]\!]/x]) \rightarrow [\![C_{\mathsf{op2}}]\!][\widetilde{A/X}][[\![v]\!]/x] <: \forall\alpha.[\![\Sigma]\!] \rightarrow ((y : [\![T]\!]) \rightarrow [\![C_1]\!]) \rightarrow \tau'$.

Then by Lemma 47 with (ii), we have

- $\Gamma \vdash \forall\alpha.[\![\Sigma]\!] \rightarrow ((y_{\mathsf{op}} : [\![T_{\mathsf{op2}}]\!][\widetilde{A/X}][[\![v]\!]/x]) \rightarrow [\![C_{\mathsf{op1}}]\!][\widetilde{A/X}][[\![v]\!]/x]) \rightarrow [\![C_{\mathsf{op2}}]\!][\widetilde{A/X}][[\![v]\!]/x] <: \tau$,

that is,

$$\Gamma \vdash [\![\Sigma \triangleright T_{\mathsf{op2}}[\widetilde{A/X}][v/x] \ / \ (\forall y_{\mathsf{op}}.C_{\mathsf{op1}}[\widetilde{A/X}][v/x]) \Rightarrow C_{\mathsf{op2}}[\widetilde{A/X}][v/x]]\!] <: \tau \ .$$

Now we have the conclusion with $C = \Sigma \triangleright T_{\mathsf{op2}}[\widetilde{A/X}][v/x] \ / \ (\forall y_{\mathsf{op}}.C_{\mathsf{op1}}[\widetilde{A/X}][v/x]) \Rightarrow C_{\mathsf{op2}}[\widetilde{A/X}][v/x]$.

Case $c = (\textbf{with } h \textbf{ handle } c)^C$: We have $[\![c]\!] = [\![c]\!] \ [\![C]\!] \ [\![h^{ops}]\!] \ [\![h^{ret}]\!]$ where

$$\begin{cases} h = \{\textbf{return } x_r^{T_r} \mapsto c_r, (\mathsf{op}_i^{\widetilde{X_i:B_i}}(x_i^{T_{i1}}, k_i^{(y_i:T_{i2}) \rightarrow C_{i1}}) \mapsto c_i)_i\} \\ [\![h^{ret}]\!] = \lambda x_r : [\![T_r]\!].[\![c_r]\!] \\ [\![h^{ops}]\!] = \{(\mathsf{op}_i = \Lambda \widetilde{X_i : \widetilde{B}_i}.\lambda x_i : [\![T_{i1}]\!].\lambda k_i : (y_i : [\![T_{i2}]\!]) \rightarrow [\![C_{i1}]\!].[\![c_i]\!])_i\} \end{cases}$$

By Lemma 50, we have

(i) $\Gamma \vdash [\![c]\!] : \tau'$,
(ii) $\Gamma \vdash [\![h^{ops}]\!] : \tau_1$, and
(iii) $\Gamma \vdash [\![h^{ret}]\!] : \tau_2$

for some τ', τ_1 and τ_2.

By Lemma 48 with (ii) and (iii), we have

(iv) $\left(\Gamma \vdash \Lambda \widetilde{X_i : \widetilde{B}_i}.\lambda x_i : [\![T_{i1}]\!].\lambda k_i : (y_i : [\![T_{i2}]\!]) \rightarrow [\![C_{i1}]\!].[\![c_i]\!] : \tau_i\right)_i$,
(v) $\Gamma \vdash \{(\mathsf{op}_i : \tau_i)_i\} <: \tau_1$,
(vi) $\Gamma, x_r : [\![T_r]\!] \vdash [\![c_r]\!] : \tau_3$, and
(vii) $\Gamma \vdash (x_r : [\![T_r]\!]) \rightarrow \tau_3 <: \tau_2$.

Then, by the second half of Lemma 50, we have

(viii) $\Gamma \vdash \tau' <: \forall\alpha.\{(\mathsf{op}_i : \tau_i)_i\} \rightarrow ((x_r : [\![T_r]\!]) \rightarrow \tau_3) \rightarrow \tau$

where α is fresh.

By the IH of (vi), we have

(ix) $(\!|\Gamma|\!), x_r : T_r \vdash c_r : C_r$ and
(x) $\Gamma, x_r : [\![T_r]\!] \vdash [\![C_r]\!] <: \tau_3$

for some C_r.

By repeatedly inverting (iv) and by Lemma 47, we have

(xi) $\left(\Gamma, \widetilde{X_i : \widetilde{B}_i}, x_i : [\![T_{i1}]\!], k_i : (y_i : [\![T_{i2}]\!]) \rightarrow [\![C_{i1}]\!] \vdash [\![c_i]\!] : \tau_i'\right)_i$ and
(xii) $\left(\Gamma \vdash \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : [\![T_{i1}]\!]) \rightarrow ((y_i : [\![T_{i2}]\!]) \rightarrow [\![C_{i1}]\!]) \rightarrow \tau_i' <: \tau_i\right)_i$

for some τ_i'. By the IH of (xi), we have

(xiii) $\left((\!|\Gamma|\!), \widetilde{X_i : \widetilde{B}_i}, x_i : T_{i1}, k_i : (y_i : T_{i2}) \rightarrow C_{i1} \vdash c_i : C_i\right)_i$ and
(xiv) $\left(\Gamma, \widetilde{X_i : \widetilde{B}_i}, x_i : [\![T_{i1}]\!], k_i : (y_i : [\![T_{i2}]\!]) \rightarrow [\![C_{i1}]\!] \vdash [\![C_i]\!] <: \tau_i'\right)_i$

for some C_i's. By (Sc-Fun) and (Sc-PPoly) with (xiv), we have

- $\left(\Gamma \vdash \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : [\![T_{i1}]\!]) \rightarrow ((y_i : [\![T_{i2}]\!]) \rightarrow [\![C_{i1}]\!]) \rightarrow [\![C_i]\!] <: \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : [\![T_{i1}]\!]) \rightarrow ((y_i : [\![T_{i2}]\!]) \rightarrow [\![C_{i1}]\!]) \rightarrow \tau_i'\right)_i$.

Then, by Lemma 47 with (xii), we have

(xv) $\left(\Gamma \vdash \forall \widetilde{X_i : \widetilde{B}_i}.(x_i : [\![T_{i1}]\!]) \rightarrow ((y_i : [\![T_{i2}]\!]) \rightarrow [\![C_{i1}]\!]) \rightarrow [\![C_i]\!] <: \tau_i\right)_i$.

Thus, by Lemma 47 and subtyping with (viii), (x) and (xv), we have

- $\Gamma \vdash \tau' <: \forall\alpha.\tau_s \rightarrow ((x_r : [\![T_r]\!]) \rightarrow [\![C_r]\!]) \rightarrow \tau$

where $\tau_s \stackrel{\text{def}}{=} \{(\mathsf{op}_i : \forall \widetilde{X_i : \widetilde{B_i}}.(x_i : [\![T_{i1}]\!]) \to ((y_i : [\![T_{i2}]\!]) \to [\![C_{i1}]\!]) \to [\![C_i]\!])_i\}$. Here, we define Σ to be $\{(\mathsf{op}_i : \forall X_i : \widetilde{B_i}.(x_i : T_{i1}) \to ((y_i : T_{i2}) \to C_{i1}) \to C_i)_i\}$, Then, it holds that $\tau_s = [\![\Sigma]\!]$. That is, we have

(xvi) $\Gamma \vdash \tau' <: \forall \alpha.[\![\Sigma]\!] \to ((x_r : [\![T_r]\!]) \to [\![C_r]\!]) \to \tau$.

On the other hand, by the IH of (i), we have

(xvii) $(\!|\Gamma|\!) \vdash c : C_0$ and

(xviii) $\Gamma \vdash [\![C_0]\!] <: \tau'$

for some C_0. By Lemma 64, w.l.o.g., we can assume that $C_0 = \Sigma_0 \triangleright T_0 \;/\; (\forall x_0.C_{01}) \Rightarrow C_{02}$. Then, by Lemma 47 with (xvi) and (xviii), we have

- $\Gamma \vdash \forall \beta.[\![\Sigma_0]\!] \to ((x_0 : [\![T_0]\!]) \to [\![C_{01}]\!]) \to C_{02} <: \forall \alpha.[\![\Sigma]\!] \to ((x_r : [\![T_r]\!]) \to [\![C_r]\!]) \to \tau$.

Then, by inversion, we have

- $x_0 = x_r$,
- $\Gamma, \alpha \vdash [\![\Sigma]\!] <: [\![\Sigma_0]\!]$,
- $\Gamma, \alpha, h : [\![\Sigma]\!] \vdash [\![T_0]\!] <: [\![T_r]\!]$,
- $\Gamma, \alpha, h : [\![\Sigma]\!], x_r : [\![T_0]\!] \vdash [\![C_r]\!] <: [\![C_{01}]\!]$,

and

(xix) $\Gamma, \alpha, h : [\![\Sigma]\!], k : (x_r : [\![T_0]\!]) \to [\![C_{01}]\!] \vdash [\![C_{02}]\!] <: \tau$.

By Lemma 39, we have

- $\Gamma, \alpha \vdash [\![\Sigma]\!] <: [\![\Sigma_0]\!]$,
- $\Gamma, \alpha \vdash [\![T_0]\!] <: [\![T_r]\!]$, and
- $\Gamma, \alpha, x_r : [\![T_0]\!] \vdash [\![C_r]\!] <: [\![C_{01}]\!]$.

Then, by 66, we have

- $(\!|\Gamma|\!) \vdash \Sigma <: \Sigma_0$,
- $(\!|\Gamma|\!) \vdash T_0 <: T_r$, and
- $(\!|\Gamma|\!), x_r : T_0 \vdash C_r <: C_{01}$.

Therefore, by subsumption on (xvii), we have

(xx) $(\!|\Gamma|\!) \vdash c : \Sigma \triangleright T_r \;/\; (\forall x_r.C_r) \Rightarrow C_{02}$.

Thus, by (T-Hndl) with (ix), (xiii) and (xx), we have

$$(\!|\Gamma|\!) \vdash \textbf{with}\ h\ \textbf{handle}\ c : C_{02} \ .$$

Also, by Lemma 39 and 40 with (xix), we have

$$\Gamma \vdash [\![C_{02}]\!] <: \tau \ .$$

Now we have the conclusion with $C = C_{02}$.

□

Corollary 68 (Backward type preservation (for closed expressions))**.**

- *If $\emptyset \vdash [\![v]\!] : \tau$, then there exists some T such that $\emptyset \vdash v : T$ and $\emptyset \vdash [\![T]\!] <: \tau$.*
- *If $\emptyset \vdash [\![c]\!] : \tau$, then there exists some C such that $\emptyset \vdash c : C$ and $\emptyset \vdash [\![C]\!] <: \tau$.*

Proof. Immediate from Theorem 67 since $\emptyset$ is obviously cps-wellformed. □

Received 2023-07-11; accepted 2023-11-07